A DISTRIBUTED CONTROL RECONFIGURATION ALGORITHM FOR 2-DIMENSIONAL MESH ARCHITECTURES WHICH TOLERATES SINGLE FAULTS PER ROW

by

Tennis S. White

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

APPROVED:

Dr. F. G. Gray, Chairman

Dr. J. G. Tront

Dr. J. C. McKeeman

May, 1988

Blacksburg, Virginia

ABSTRACT

A reconfiguration is developed for 2-dimensional mesh architectures and applied to a fault tolerant cellular architecture. The reconfiguration is accomplished by adding communications paths to each cell which can be enabled by means of transistor switches controlled by decoding the contents of a register containing the relative position of faulty cells. This enables faulty cells to be bypassed and operations of cells in the same row east of the faulty cell to be shifted one cell to the east and a spare cell included in the active pattern.

A modified s-value algorithm is also developed which enables a cell to determine the size of a square pattern that may be centered on that cell.

ACKNOWLEDGMENTS

The author wishes to thank the chairman of his committee, Dr. F. G. Gray, for his inspiration, guidance and assistance over the past three years, and the other members of the committee, Dr. J. G. Tront and Dr. J. C. Mckeeman for their advice and assistance on this thesis.

Thanks also go to Dr. J. Cleary and Dr. R. Baker of Research Triangle Institute, Research Triangle Park, N. C. for their financial support. Their assistance over the last two years is greatly appreciated.

Special thanks go to my wife, , without whose help this could not have been possible, and to my children and .

Table of Contents

List of Illustrations

1.0 INTRODUCTION

1.1 THE SYSTOLIC ARRAY

A systolic array as defined by H. T. Kung [1] is ideally constructed using a few simple processing elements(PE's or cells) with a simple interconnect configuration, that greatly increases computational power through concurrent computation [1] [2] and pipelining [3]. An algorithm is broken into many small blocks, with each PE executing different parts of the algorithm. Data is transferred to the array by the host machine in a regular rhythmic rate. Each data item is used by a cell and then passed to a neighboring cell for further computation [3] [4]. Using current technology, it is believed that any significant increase in computational power must come from the use of systolic architectures [5].

1.1.1 ADVANTAGES OF SYSTOLIC ARCHITECTURES

The advantages of systolic arrays are many. The simple structure, in which many copies of a few cell types are placed on a chip, makes the array ideal for Very Large Scale

Integration(VLSI) and Wafer Scale Integration(WSI) [6] [7]. The large size of these chips decreases both the propagation delays and the power consumption [8] [9]. Since each operand is used in multiple computations, the systolic array is an excellent solution to computation-bound problems [10] [1]. The amount of memory accesses is greatly decreased [1] [11], thus decreasing the load on system communications.

The use of multiple processors for execution gives speed-up proportional to the number of processing elements [10]. Communications within the array can be optimized to the specific algorithm under consideration [12] [13] [4] [14] [7]. This has a double effect since speed is increased, and power consumption due to communications is decreased. Perhaps the most important advantage is the ability to decrease the load on the host machine by implementing in hardware parallel computations which were previously done serially with software [15].

1.1.2 DISADVANTAGES OF SYSTOLIC ARRAYS

Some of the items considered to be advantages of systolic arrays lead directly to some of the most costly disadvantages. The optimal communications network for one algorithm is seldom optimal for others. Since the network is usually fixed [16] [4] [17] [7], it may not be optimal for more than one algorithm [18]. Attempts to execute an algorithm on an array with less than the optimal communications will lead to decreased computational speed, since clocks must be slowed to allow for long path problems [15].

The array is algorithm specific, and therefore has limited use [19] [7]. This increases the per-unit cost of the array since the most of the chip cost is in the design [20] and not in manufacturing. The high cost can further limit usability [1]. This cost is increased further by manufacturing and operational faults. A fault in a single cell of the array frequently leads to

total failure of the chip. [8] [1]. This problem is magnified by large scale integration, since the increased size leads to the higher probability of faults [9] [21] [22].

Many algorithms have already been developed for implementation on systolic architectures. These include matrix algebra applications [1] [2] [11] [23]-[38], signal and image processing [25] [39]-[54], and fourier transforms [1] [5] [55] [56]. Also implemented are applications in graph theory [40] [57]- [60], least-squares algorithms [61] [62], integer arithmetic [63]-[66] and polynomial solutions [67] [68]. Algorithms also exist for solutions to problems in beam-forming [40] [44] [69], digital filtering [70], gausian elimination [71], medical sciences [72], dictionary machines [73], solutions to linear equations [74] and problems in finite fields [75].

1.1.3 FAULT TOLERANCE REQUIREMENTS

In systems where the cost of failure is high, it is important that fault tolerance be an integral part of the design. Steps must be taken to insure that faults are detected isolated. Available methods for achieving this are masking redundancy, concurrent error detection, hardware redundancy, time redundancy, algorithm modification or reconfiguration [76].

If the systolic array is to become a feasible product, ways must be found to make use of its advantages, while decreasing the effect of the disadvantages, and adding a high level of fault tolerance. Reconfiguration can increase the through-put of an array, by adapting to the ideal communications configuration, and decrease the cost by adding fault tolerance [22], which increases both manufacturing yields and expected lifetime of the array.

It is desirable to be able to reconfigure wafer scale architectures to adopt many algorithmically specialized configurations [22]. However, any reconfiguration algorithm used must insure that minimum communications path length is used [77]. Existence of long communi-

cations paths in an array increase both the propagation delays and the chip failure probability [78].

Reconfiguration is available in three ways: [79]

1. Static reconfiguration can be used during fabrication processes to bypass faulty cells. This offers increased manufacturing yields, but does nothing to increase tolerance to operational faults or allow for the execution of different algorithms.

2. Dynamic reconfiguration controlled by the host machine can be used to increase the yield and configure for different algorithms, but external control of the configuration process or the fault detection process is not desired [8]. This results from latency problems encountered when both the results of external testing and reconfiguration commands are delayed by long communications delays.

3. Dynamic reconfiguration under local control, or control of the array itself has the advantages of the above scheme, but eliminates the latency problems since testing and reconfiguration are local functions. [8] This method is desirable even though there is increased hardware cost to enable testing and reconfiguration.

1.2 RECONFIGURATION TECHNIQUES AVAILABLE

Several techniques currently exist for the purposes of algorithm and fault- tolerance reconfiguration. For the most part, these require the addition of transistor switch networks to the inputs and/or outputs of the processing elements. The techniques can be divided into two types, external control and local control.

1.2.1 ALGORITHMS WITH DISTRIBUTED CONTROL

There are four types of algorithms which offer local control of the reconfiguration process. Those are the message-passing [14], request/response [80], programmable-links [81], the programmable- multiplexer [82] [79] [83]-[85], and the local supervisor [86].

The message-passing types append routing information to the actual data being transmitted. Each cell is equipped with a communications processor [14] that examines the received message, and if the current cell is the destination, the data is passed to the cell. If not, the message is modified and routed to the next cell in the network. Each communications processor has access to many communications links, and uses an elaborate algorithm to determine what links have not been used by other cells, and the routes available to transmit data to the receiving cell. This process has the disadvantage of increased delay because of processing necessary by intermediate communications processors [14].

The request/response algorithm [80] is implemented by connecting each cell to each of its six neighbors to the east and west. Each cell transmits a request to enable communications to each of its neighbors to the west. When the cells on the western boundary of the array receive a request, they choose the one with the highest priority and transmit a response. Cells that receive a response from the cells to the west then choose the highest priority response and complete the communications link. There may be several iterations required before the final configuration of the array is completed. Information about the location of faulty cells is transmitted throughout the array to prevent dead-end connections between cells that cannot complete a full-length row or column. Fault-free cells that become part of a full row then complete connections to form columns by enabling connections to a vertical bus. The disadvantages of this algorithm are the number of iterations required to complete the rows, and the existence of a long bus to form the vertical connections between cells.

The multiplexed algorithms [82] [79] [83]-[85] use signals generated by each cell defining their condition, whether faulty or good. This signal is used, together with combinations of the same signal from cells in neighboring rows and to the left, to determine which of several input signals to the multiplexer will be routed into a cell, or if any data will be routed to the current cell. Depending upon the size and number of multiplexers added, this algorithm can achieve a very large degree of fault tolerance. In some cases however, they will use fault-free cells as data transmitters, without using the cells computational capabilities. They have the disadvantage of requiring communications of the control signals across the entire length, and sometimes width of the array. This leads to increased failure probability [78].

The programmable-link [81] algorithm is similar to the above algorithms in that each cell is required to transmit a fault-free signal defining its current condition. These signals are used to control transistor switches into and out of communications links.

The algorithm of Yanney and Hayes[86] offers local control of the reconfiguration process but has some minor problems. The algorithm executes reconfiguration by assigning each cell a unique state. Each state has the responsibility of determining the availability of all other cells in its neighborhood, including spares. A cell must also have knowledge of the connectivity of all cells in its neighborhood. When a cell is determined to be faulty, its state is set to -1. When the cell responsible for recovering from the loss of that cell determines that state is not available, it searches for a spare cell with connections that allow it to be substituted directly for the faulty cell. If it finds one, it changes the state of the spare cell. If it does not find a spare, the responsible cell assumes the state of the missing cell, and another cell tries to find a replacement for the original supervisory cell. This process continues until the entire pattern is regenerated. No discussion is made on the control of the communications network.

1.2.2 CENTRALIZED CONTROL ALGORITHMS

Algorithms that use centralized control of the configuration process and cell test cost less in hardware [8] than those using distributed control and testing. Distributed testing, where cells test each of their neighbors, and control of the configuration process, must be used if fault tolerance is to be achieved [87]. The algorithms will be discussed briefly however.

Two of the external control algorithms are implemented by the use of large crossbar networks [88] or by connecting two cells through buss bars [21]. Control signals from the host machine are used to control sets of transistor switches to enable the necessary connections. Another algorithm accomplishes the reconfiguration by routing signal paths around faulty cells by the use of bypass circuits [89]. This is an effective reconfiguration scheme, but introduces undesirable delay through the additional circuitry and should not be used [90].

The diogenes reconfiguration algorithm controls the configuration of the array by adding a network of latch controlled switches to the input of each array [87]. The number of switches added depends upon the amount of fault tolerance desired. The primary purpose of this technique is to maintain the maximum number of full rows or columns in the array. This algorithm has the disadvantage of requiring long bus lines for the control signals [14].

The redundancy algorithm [87] adds a "spare" row of cells to an array. The location of the spare cells is continuously shifted throughout the array, with the current spare cell's inputs configured to be identical to an adjacent cell in the same column. The function of the spare cell is to duplicate the computations of the adjacent cell, and if an error is detected, the spare cell assumes the responsibilities of the faulty cell. The implementation of this algorithm requires the addition of a large number of circuits to be able to compare the outputs of all pairs of adjacent cells (Figure 1 on page 9). The problems with this algorithm are long communications paths required to configure around the spare cell locations, and high fault latency.

The location of a faulty cell cannot be determined until it is tested by the spare cell. This may be a problem, especially in very large arrays.

The covering approach [91] adds both a spare row and a spare column to an array, along with an external controller. Reconfiguration is achieved by routing data to fault-free cells through an extensive network of busses and switches. This network of busses and the external controller create less than maximal fault-tolerant conditions.

1.2.2.1 *THE DYNAMICALLY CONFIGURABLE COMPUTER*

The dynamically configurable computer array [7] offers a solution to most of the disadvantages of systolic arrays. This architecture is composed of polymorphic [7] processing elements, capable of executing more than one type of computation. These cells are connected via a communications network consisting of many links controlled by programmable switches. Both the processing elements and the switches are under control of an external program, so the function of each processing element, and its communications functions with other cells within the array can be modified to the optimal configuration of the current execution algorithm.

The array also offers a high degree of fault tolerance, both during the manufacturing process and normal operations [7] since the switches can be used to isolate faulty cells. This is especially necessary in WSI [22] since the lack of fault tolerance can be very expensive in discarded parts.

The major disadvantage of this architecture is its requirement for an external controller for test and configuration. This creates difficulty in generating effective testing, since all points of the internal array cannot be sampled [87], and creates long communications lines internally between the switches, cells and the external controller.

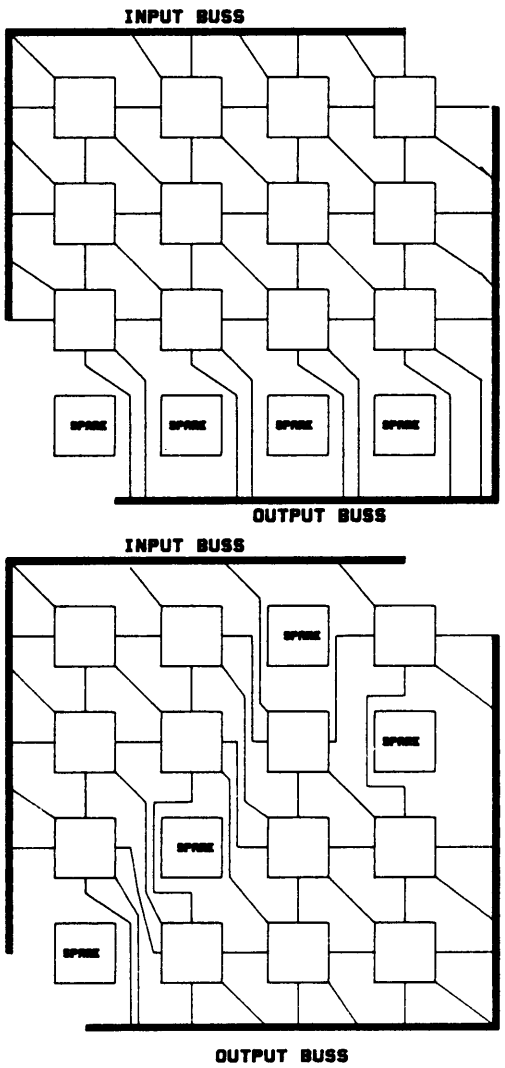


Figure 1. Redundant Approach: Array Showing Initial Spare Row and Shifted Spares

1.2.3 THE FAULT-TOLERANT CELLULAR ARRAY

A solution to the above problem is a cellular architecture [92] [94] (Figure 2 on page 11) in which each cell or processing element of the array is composed of independent control and execution sections. The execution plane of the array is responsible for algorithmic computations, and for maintaining the communications channels required with other cells within. A processor-switch [7] (Figure 3 on page 12) lattice serves this purpose very well, with each cell having sufficient computational power to select primary and alternate paths through programmable switches [7] to other cells required to execute the current algorithm. The information needed to determine the exact function of each cell, and the interconnection requirements, would be communicated to the cells of the execution plane from their associated control cells.

1.2.3.1 THE CONTROL PLANE

The control plane is composed of cells that require less computational power than that of the execution cells, and have a less complicated two- dimensional nearest-neighbor mesh [93] (Figure 4 on page 13.) communication network. This system is sufficient to enable each cell to determine its next state, and therefore the next state of the execution cell, by examining its current state and the current state of its four nearest neighbors. The state of a cell is defined as the computation operation to be performed.

Communications channels in the control plane are serial and multiplexed, with only one channel existing between each pair of cells. For this reason each cell is assigned a priority which determines whether the cell receives or transmits during any particular clock period. All cells with a priority of 1 communicate only with cells with priority 2, and vice- versa.

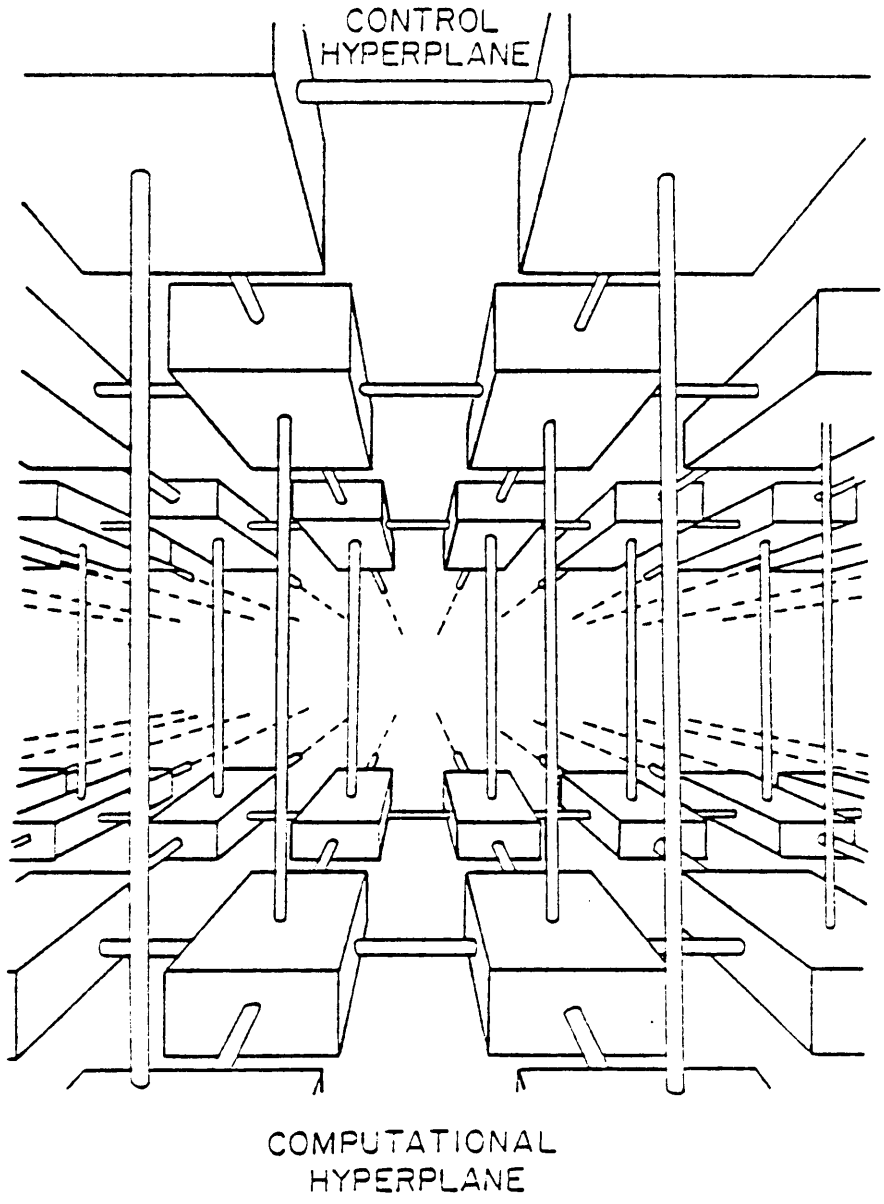


Figure 2. Cellular Array

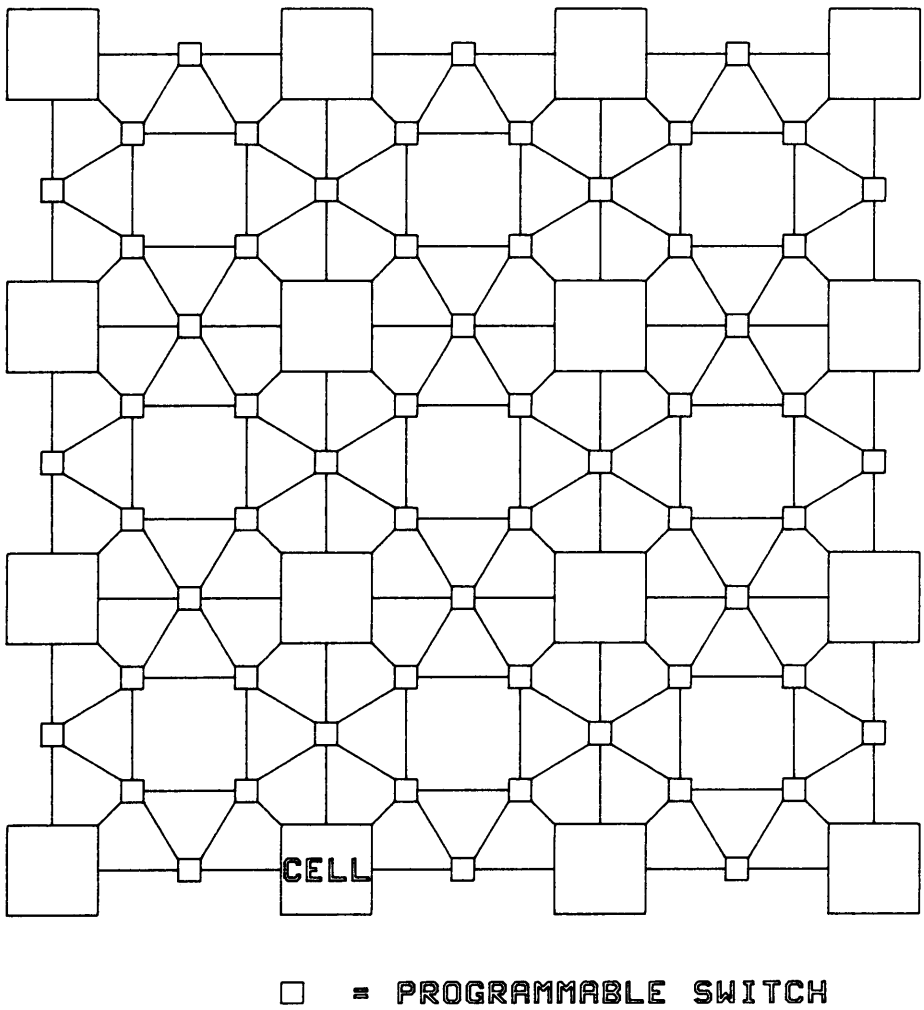


Figure 3. Processor Switch Array:

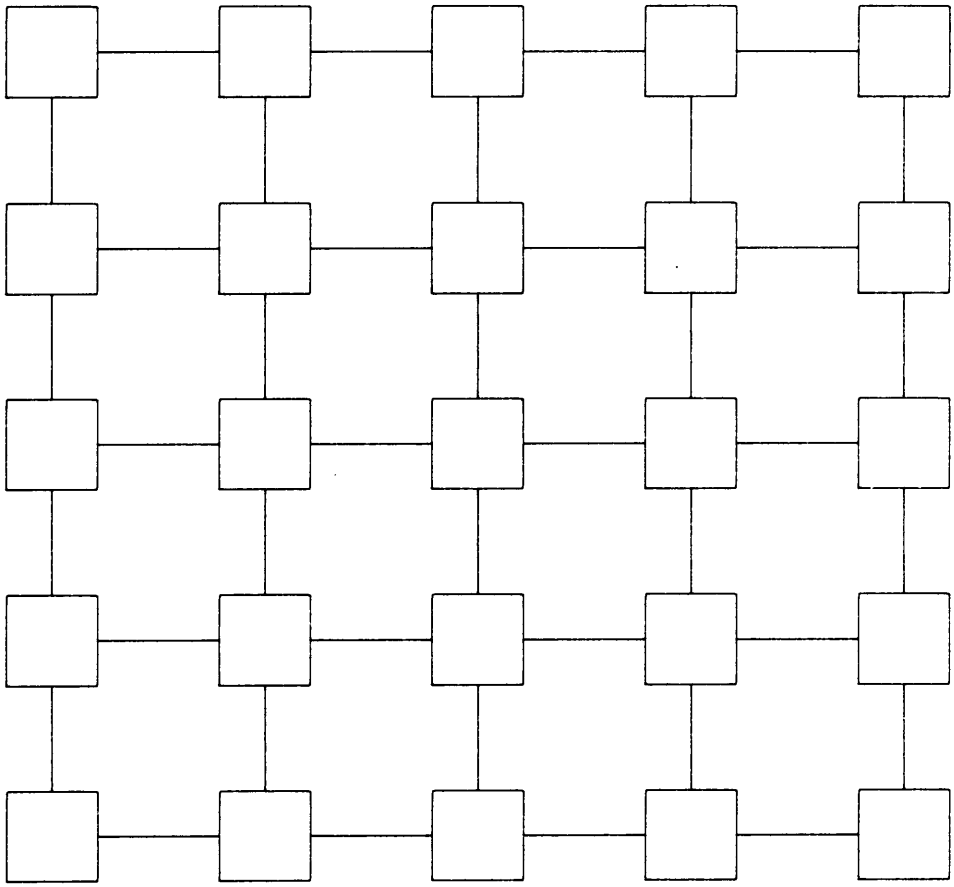


Figure 4. Cellular Array: Showing Nearest Neighbor Mesh Interconnection

To be able to execute many algorithms efficiently, each cell in the control plane must be able to determine the size of the fault-free area surrounding it. This information is stored in a register contained in each cell called the s_value [92]. The s_value of a cell is determined by reading the s_value of each of the neighboring cells and incrementing the minimum value read by one. A cell with an s_value of three has a minimum of three fault-free cells in all directions, north, south, east or west. Cells that form the outside boundaries of the matrix are given an s_value of zero. Cells used to isolate faulty regions of the matrix are given an s-value of -1, and are not allowed to participate further in the operations of the matrix.

The clocking of the system is in six phases. The first three periods are dedicated to calculation of the s_value, the remaining three to generation of next state information in the control plane. During clock period one, cells with priority one transmit their current s_values. Cells with a priority of 2 are placed in the receive mode. In phase two, the operations are reversed. Phase three is used by all cells to calculate new s_values. This value becomes the new s-value of each cell. Each cell internal to the matrix will calculate a higher s- value, with cells at the center of a fault-free matrix calculating a maximum value. The s-value distribution in a fault-free array might be

0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	0
0	1	2	2	2	2	2	1	0
0	1	2	3	3	3	2	1	0
0	1	2	3	4	3	2	1	0
0	1	2	3	3	3	2	1	0
0	1	2	2	2	2	2	1	0
0	1	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0	0

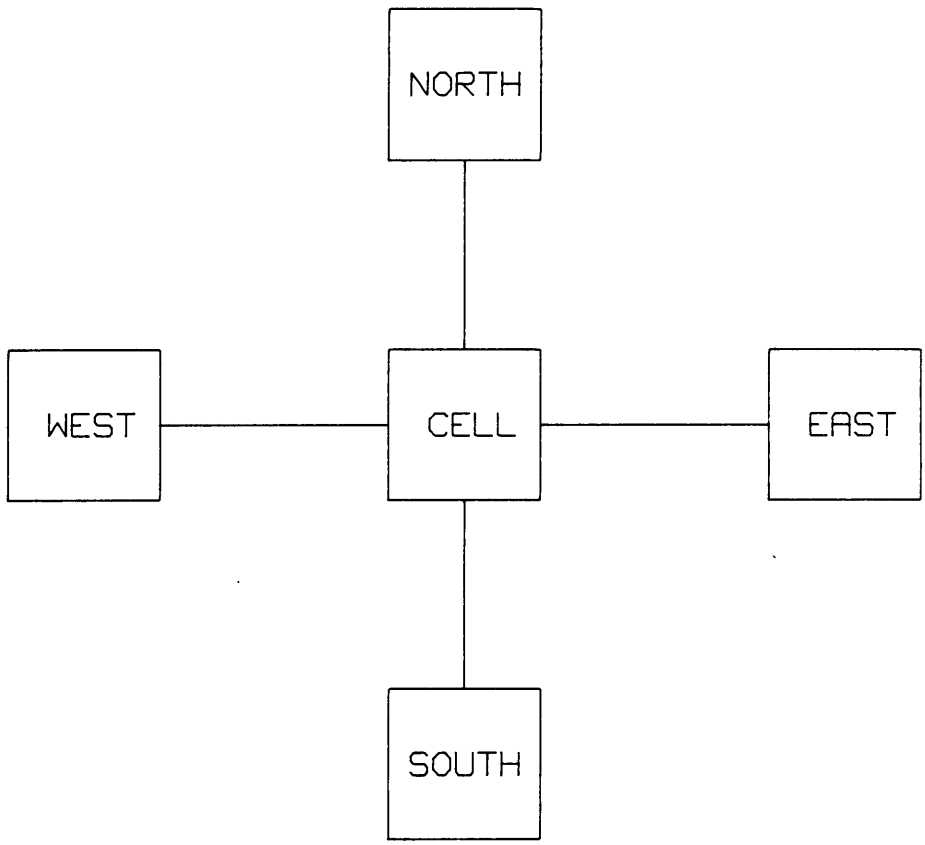


Figure 5. Nearest Neighbor Index: Showing North, South, East, and West Neighbors

A cells s-value determines the maximum dimensions of a pattern that may be generated through communications with the four nearest neighbors. Since each cell communicates only with cells in the north, south, east and west directions (Figure 5 on page 15), the pattern grown with this communications scheme will expand in a diamond shape. Other interconnection schemes would offer more flexibility in the pattern growth, but the logic required to store the required information, and determine a cells next state, would be prohibitive. Many usable configurations can be generated using this mechanism, but there are some inefficiencies. Cells on the external boundaries of the matrix can seldom be included in a computational algorithm, since patterns are grown in a diamond shape, and indeed the s- value of cells internal to the array are not affected if cells in the outside corners of the matrix become faulty. An s-value of four can also be calculated with an array of this configuration.

```
            0
          0 1 0
        0 1 2 1 0
      0 1 2 3 2 1 0
    0 1 2 3 4 3 2 1 0
      0 1 2 3 2 1 0
        0 1 2 1 0
          0 1 0
            0
```

The first array would support a square matrix of seven by seven, while the second will support only a three by three square, if those with an s_value of zero are not included.

1.2.3.2 PATTERN GROWTH

The array pattern is developed in a fault-free area of the existing matrix through a process known as pattern growth or generation. A seed [92] is deposited in a fault-free cell either from an external source, or generated by internal logic. The value of the seed is sufficient for the cell containing the seed to determine the exact pattern to be generated, and the number of cells required in the pattern. The cell containing the seed examines the size of its s-value, and if the adjacent fault-free area is sufficient, pattern generation commences. If not, the seed is passed to a neighboring cell.

This process takes place during phases four, five and six of the clock. During phase four of the clock, priority one cells transmit their current state and priority two cells receive. At the next clock pulse, the operations are reversed and during phase six, each cell calculates its next state. The pattern growth might proceed in the following steps.

0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0
0 0 0 0 0	0 0 b 0 0	0 a b c 0	0 A B C 0
0 0 s 0 0	0 c d e 0	0 d e f 0	0 D E F 0
0 0 0 0 0	0 0 f 0 0	0 g h j 0	0 J K L 0
0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0.

This can be accomplished by designing the state logic such that a current state configuration of s0000; where the state locations are self, north, south, east and west; leads to a next state of d, and other initial configurations lead to states b, c, e and f.

Using the above processes, the cells in the control plane are able to produce a pattern of states equivalent to that needed to execute the algorithms for which the array is programmed.

1.2.3.3 *DISADVANTAGES OF THE ARCHITECTURE*

The main drawback of this array algorithm is its inability to function with faulty cells inside the pattern, or to generate a pattern around a faulty cell. Should a fault occur, either in the control plane or in the execution plane, the faulty cell must be isolated with quiescent cells, with s-values of -1, and a pattern regrowth process begun which will regenerate the same pattern in another section of the matrix which contains no faulty cells. Although the execution cells have the ability to isolate the faulty region by changing the communications configuration to isolate the faulty cell, the cells in the control plane do not have this ability. Communications within the control plane is limited to the four nearest neighbors of each cell, whether the neighbor is faulty or not. Therefore, any fault that occurs dictates that the faulty cell be isolated and the state pattern regrown in a fault-free area of the chip or wafer.

The solution to this problem and the subject of this thesis is to give cells within the control plane the ability to configure around faulty cells, without requiring that a complete pattern regeneration be accomplished. Operations previously accomplished by the faulty cell must be assumed by a cell that is aware of the function of the faulty cell. Since each of the four fault- free neighbors of the faulty cell store the state, it is logical that one of the four neighbors assumes the state of the faulty cell, and the state of that neighbor be passed to one of its neighbors. The pattern of states in a row or column could then be shifted one cell until a spare cell at the end of the row or column is made an active member of the pattern. This will allow regeneration of the state pattern much faster than is possible with the current architecture, and make much more efficient use of fault-free cells by isolating faulty cells with switches instead of cells.

2.0 A NEW FAULT-TOLERANT INTERCONNECT ALGORITHM

This algorithm was developed with the following assumptions made.

1. A faulty cell can not be relied upon to produce any valid output signals. For this reason, methods must be found to isolate the faulty cell without utilizing fault-free cells solely for this purpose. A switch controlled communications scheme can serve this purpose.

2. Wrap-around communications will not be used within the array. It is felt that this adds communications paths to the array that severely limit array timing, especially one of wafer-scale size. Therefore wrap-around communications is not considered.

3. The array implementing this algorithm will have at least one column of spare cells on the far eastern boundary.

4. All cell faults are permanent. No provision is made for intermittent faults, although a slight modification to the algorithm would cover this possibility. Retesting a faulty cell and re-enabling the communications network should be investigated.

5. All cell failures are independent. No single cell failure will cause other cells to fail.

6. Each cell tests all cells with which it has active communications.

7. Testing will be sufficient to determine whether a fault is internal to a cell, or in the communications network. In the event of a communications network failure, the testing algorithm must determine which cell to isolate.

8. The algorithm can be implemented with the existing communications scheme in the control plane. The only change to inter-cell communications will be the number of bits transmitted between cells.

9. Communications between two cells within the array will be under the complete control of the two cells involved. No control signals will be generated, and no communications paths will be added, outside cell boundaries.

10. All four cells communicating with a faulty cell will detect the fault during the same clock period.

11. The cells in the control plane are used only to determine operations of execution cells. Programmable switches in the execution plane are either controlled by the execution cells, or are controlled by control cells in another plane.

2.1 THEORY OF OPERATION

Each cell in the normal two-dimensional-mesh communicates with only the four nearest neighbors (Figure 4 on page 13). This algorithm requires the addition of connections to the

northeast, northwest, southeast and southwest cells, as well as connections to two additional cells within the same row (Figure 6 on page 22). These connections give each cell communications with the two cells to the east and two cells to the west , plus the six cells in the northern and southern rows for a total of ten. The communications paths to the north, northeast, northwest, south, southeast and southwest are required to enable the array to maintain a maximum number of full length columns. The extra paths to the east and west will allow reconfiguration to complete communications within a row, although the width of a row will be reduced by one when the first fault occurs. The second and all succeeding faulty cells within a row will cause communications along the rows and columns to be interrupted, and the rows and columns will be segmented.

The means of controlling the communications paths can be provided by adding to each cell, a ten bit register that will contain the relative location of faulty cells in each cells own row, the row to the north and the row to the south. Sufficient logic must also be added to each cell to receive and transmit these bits , and control the setting of the bits in the fault-register. The bits of the fault-register can be decoded by a Programmed-Logic-Array(PLA), with the decoded signals controlling the communications network through pass-transistors added to each path. Since there will be pass-transistors on each end of a path, two cells must be attempting communication with each other before a path is complete. The communications can be broken by either of the two cells. This will be an effective method of isolating faulty cells.

For proper implementation of this algorithm, data pertaining to the fault condition surrounding any cell must be transmitted to other cells. Assuming the array is designed to separate the computation and testing cycles, it is possible that the fault data could be communicated either as part of normal data communications, or as a portion of the data transferred during the testing cycles. The method selected in any implementation should be the one that creates the smallest time penalty or fault latency.

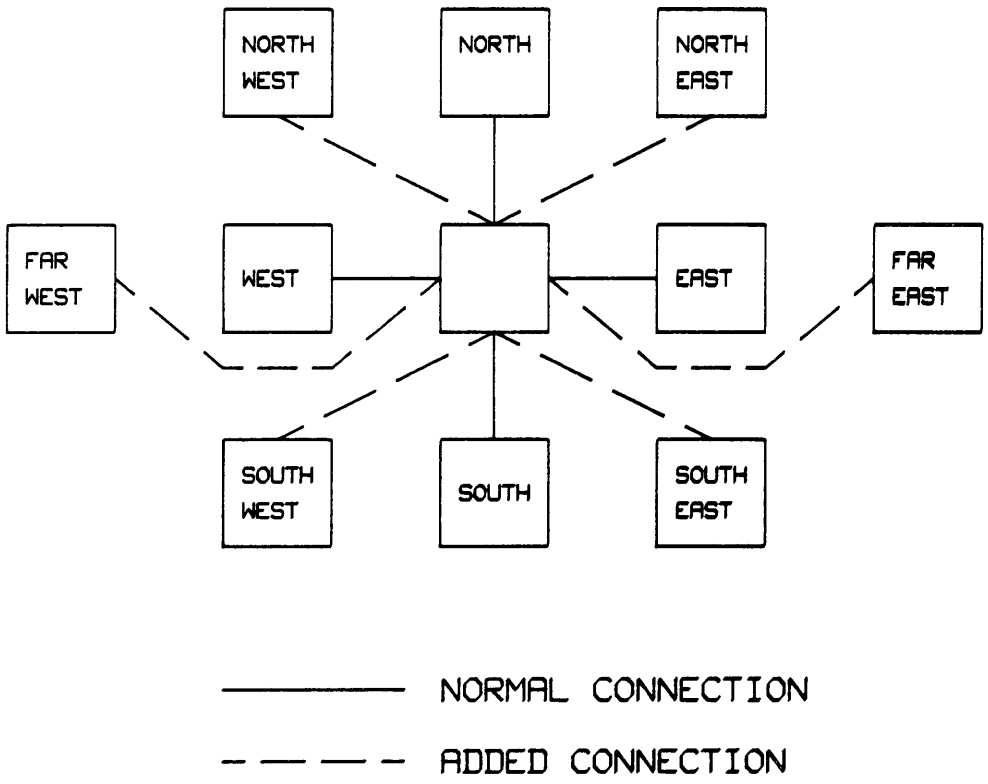


Figure 6. New Neighborhood Index: Showing Added Interconnections

The following is a list of the bits added to implement this algorithm, and their logical meaning.

BIT NO.	FAULT LOCATION
1	WEST CELL FAULTY
2	NORTH CELL FAULTY
3	SOUTH CELL FAULTY
4	ANY FAULT TO THE WEST OF THE WEST CELL(FAR-WEST)
5	ANY FAULT TO THE WEST OF THE NORTH CELL(NORTHWEST)
6	ANY FAULT TO THE WEST OF THE SOUTH CELL(SOUTHWEST)
7	EAST CELL IS FAULTY
8	ANY FAULT TO THE EAST OF THE EAST CELL(FAR-EAST)
9	ANY FAULT TO THE EAST OF THE NORTH CELL(NORTHEAST)
10	ANY FAULT TO THE EAST OF THE SOUTH CELL(SOUTHEAST)

The bits in the register are set in the following manner.

1. Bit one is set only if a cell detects a faulty cell on its communications path to the western cell.

2. Bit two is set if

   - A cell detects a faulty cell on its communications path to the northern cell.

   - The input from the northeastern cell contains bit one.

   - The fault information from the northwestern cell contains bit seven as a logical one. This is done to communicate to a cell, the existence of a faulty cell to its immediate north or south, when the two cells are not connected. If a fault has previously occurred in a row, cells to the east of the fault may not be in communications with cells to their north. If a cell to the north or south becomes faulty, an extra major clock cycle

is saved by allowing this information to be received from other cells instead of through testing.

3. Bit three is set if a cell detects a faulty cell on its communications path to the south, or by communications about the fault similar to that for bit two.

4. Bit four is set if the cell detects a fault on its enabled connection to the far-western cell, or if its fault register input from the western cell has either bit one or bit four set.

5. Bit five is set if the cell is connected to a faulty northwest cell, or if the fault register input from either the north or the northwest cell has either bit one or bit four set.

6. Bit six is set if the cell detects a faulty cell to the southwest, or if the input from the southern or southwestern cell contains bits one or four set to a logical one.

7. Bit seven is set if a cell detects a faulty cell on its immediate eastern edge.

8. Bit eight is set if

   - A cell detects a faulty cell on its communications path to the far-eastern cell. This could occur only if the cell to the east were already faulty.

   - The input from the eastern cell contains either bit seven or bit eight set to a logical one.

   - The input from the far-eastern cell contains either bit seven or bit eight set to a logical one.

9. Bit nine is set if

   - The fault information from the northwestern cell contains bit four.

- The fault information from the northern cell contains either bit seven or bit eight.
- The fault register input from the northeastern cell contains either bit seven or bit eight.
- The input from either the eastern or the far-eastern cell contains either bit two or bit nine.

10. Bit ten of the fault register is set if

- The input from the southwestern cell contains bit eight.
- The input from the southern cell contains either bit seven or bit eight.
- The fault information from the southeastern cell contains either bit seven or bit eight.
- The input from either the eastern cell or the fareastern cell contains either bit three or bit ten.

2.1.1 SINGLE FAULTS

When the first detectable fault occurs in a cell(i,j), it is recorded in the fault register in each of the four cells with which it communicates, cell(i-1,j), cell(i,j+1), cell(i,j-1) and cell(i+1,j). These four cells immediately make a decision to reconfigure, and will change their communications connections accordingly. During the next transfer cycle , the entire contents of the fault register is placed on the output lines of the four cells. All other cells connected to these four will receive the data, analyze it and set their fault register accordingly. However, since the original cells reconfigured immediately upon detection of the fault, they will temporarily be disconnected from any cells not in their row.

The four cells detecting the fault initially will change their north-south and east-west connections so that the two cells to the east and west, cell(i,j-1) and cell(i,j+1), will enable a connection between themselves, and disable their paths to the faulty cell. The cell to the east of the faulty cell will enable its north-west and south-west ports, while the cell to the north enables its southeast port, and the cell to the south enables its northeast port. The cell to the west of the initial fault makes no changes to its north-south connections. The effect of this is to leave the cells to the north and south of the east cell with no north- south connections (Figure 7 on page 27). For this reason, the primary means of communicating the location of a faulty cell is along the east-west communications channel, with north- south being used only in the special cases listed above.

It can be clearly seen that the relative location of a faulty cell will be transmitted through two columns, one to the east and one to the west, each major clock cycle. This delay could be decreased, but only at the cost of adding additional communications paths between cells, as was the case with the controlled multiplexer algorithms. Methods must be found to notify the external, or internal source of data when the reconfiguration is complete. This could be implemented either by counting clock periods after a fault detection and allowing sufficient time for the longest reconfiguration delay to take place, or by giving the logic in the spare cell in each row the ability to flag the completion of the reconfiguration.

After sufficient time has elapsed, cells in the rows adjacent to a single faulty cell(i,j) will take on the following configuration.

1. Cells in the row to the north of the faulty cell will not modify their connection scheme if the fault cell is to the southeast. This includes cells (i-1,1) thru (i-1,j- 1) (Figure 8 on page 29).

2. If the faulty cell is directly south, or to the southwest of a cell in the northern row,(cells (i-1,j) thru (i-1,N), where N is the total number of cells in any row) each cell will modify its

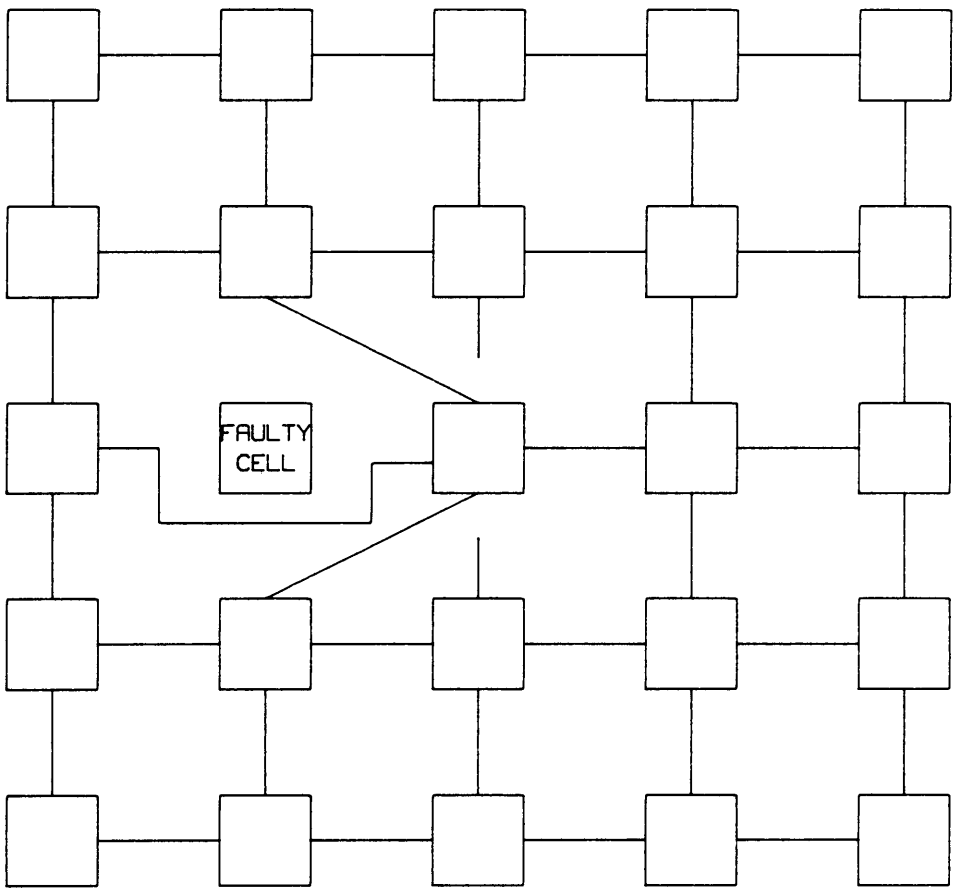


Figure 7. First Reconfiguration Step: Showing Enabled Connections Immediately After Fault Detection.

southern communication path one cell to the east. This isolates the faulty cell from the north, and brings into the active array a spare cell on the eastern boundary (Figure 8 on page 29).

3. Cells in the row to the south of the faulty cell will not change connections if the cell is to west of the faulty cell((i+1,1) thru ((i+1,j-1)). They will shift their northern communications path one cell to the east if the faulty cell is directly north, or to the northwest. This isolates the fault from the south, and completes the connection to the spare cell (Figure 8 on page 29).

4. Cells in the same row, and to the west of the faulty cell((i,1) thru (i,j-1)) will make no change to their north or south connections. The one cell(i,j-1) directly to the west of the faulty cell will disconnect its path to the faulty cell, and enable the path to the cell(i,j+1) to the east of the faulty cell (Figure 8 on page 29).

5. Cells in the same row, and east of the faulty cell((i,j+1) thru (i,N)) will modify their north and south connections one cell to the west. The cell(i,j+1) to the east of the faulty cell enables its connection to cell(i,j-1).

6. Cells on the extreme eastern boundary, in the rows to the north and south of the faulty cell will lose either a south or north communications path. These may either connected to external ports of the array, or act as boundary cells, which do not have a full complement of connections to other cells (Figure 8 on page 29)

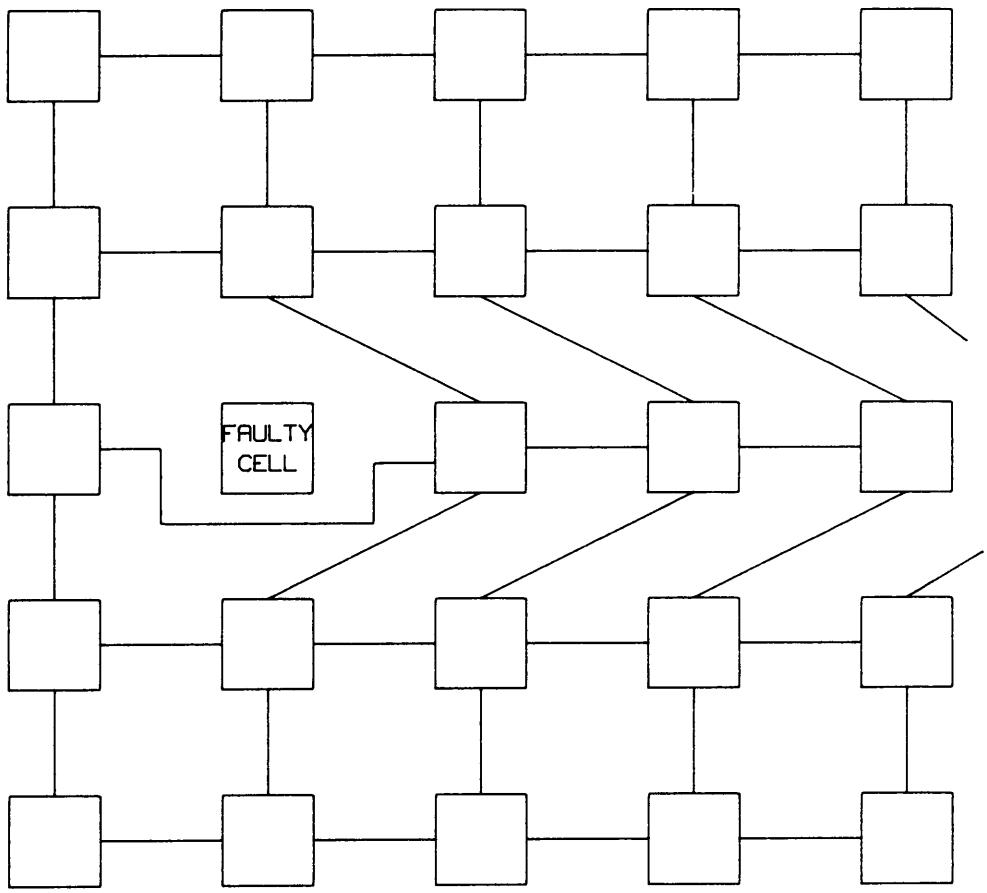


Figure 8. Single Faulty Cell: Cell Rows Showing Effect of a Single Faulty Cell

2.1.1.1 FAULTY CELLS IN ADJACENT ROWS

When faulty cells occur in two rows (Figure 9 on page 31), the reconfiguration of some of the cells can be changed. First, we assume the fault in the northern row is to the west of the fault in the southern row. When this happens, cells in both rows to the west of the faulty cells still maintain communications with cells directly north and south. If a cell in the northern row is located between the two faulty cells, or is located directly north of the faulty cell in the southern row, communications both north and south will be arranged exactly as in the case of the single faulty cell, for cells to the east of a faulty cell. If the cell in question is to the east of the faulty cell in the southern row, communications to the north remains the same, one cell to the west. However, communications to the south is now redirected to the cells southern neighbor.

Next, we consider the cells in the southern row. If the cell is west of both faulty cells, normal communications are maintained. If it is east of the northern faulty cell, or directly south of that cell, communications to the northern row is shifted one cell to the east. If the cell is east of both faulty cells, normal communications with cells directly north are maintained, while the southern paths are shifted one cell to the west.

2.1.1.2 SINGLE FAULTY CELLS IN THREE CONSECUTIVE ROWS

When faulty cells occur in three consecutive rows, the behavior of all cells matches that for faulty cells in two adjacent rows. Communications to the north for any given cell is controlled only by the relative location of faulty cells in the two rows in question. In the same manner, communications to the south for any cell is governed only by the faulty cell locations in its own row and the row to the south.

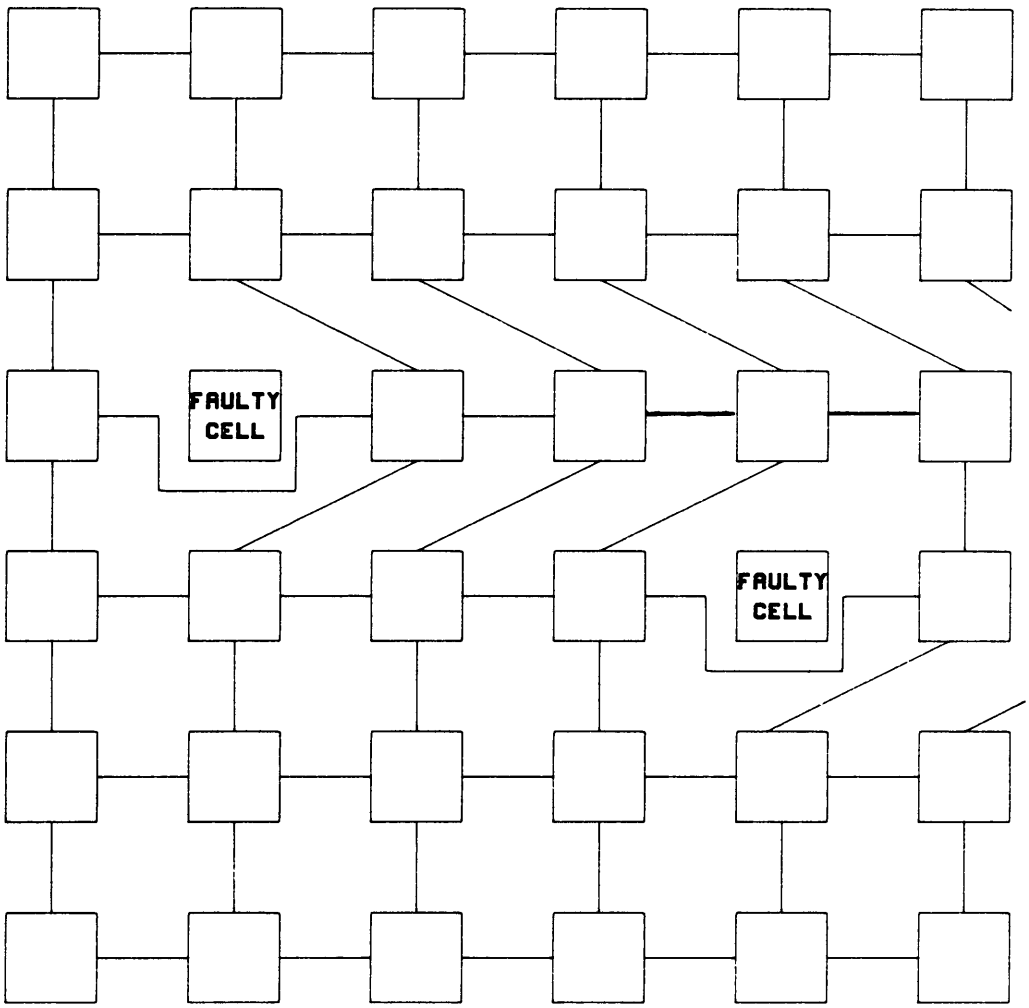


Figure 9. Two Single Faults: Effect of Single Fault in Two Adjacent Rows

2.1.1.3 *RESULTS OF SINGLE CELL FAILURES IN ANY ROW*

This interconnection reconfiguration algorithm is designed to tolerate any number of faulty cells, so long as no more than one fault occurs within any row in the array (Figure 10 on page 33). Any number of single-faults can occur within the array, and the array will have the same overall dimensions in all cases. Assuming we initially have an N by N array, the occurrence of the first faulty cell reduces the size of the array to N - 1 columns and N rows. This size will be maintained regardless of the number of single-fault occurrences. There will be segments of connected cells on the eastern border of the array, with the number and size of the segments dependent upon the number and location of the faulty cells.

2.1.2 MULTIPLE FAULT OCCURRENCES

To simplify the following discussion, faulty cells will be numbered sequentially starting with one to indicate the eastern most cell. The occurrence of the second and succeeding faults within any row in the array will reduce both the number of full length columns and the number of full width rows. The degree of reduction is dependent upon the number of multiple faults, and on their location within the array. It is possible that a double fault in each row of the array could only reduce the number of full length columns by only one, or at worst the reduction could be equal to the number of rows in the array. This would mean no full-length columns would exist if double-fault conditions existed at different locations (Figure 11 on page 34).

When multiple faults occur in any one row, reconfiguration around the easternmost fault is accomplished. Cells in the three rows west of this faulty cell take on a communications configuration that is identical to the one they would have if only the eastern fault existed. Those cells that would communicate with the other faulty cells that create the multiple fault condition lose one of their neighbors. Cells to the north of the multiple fault do not have southern

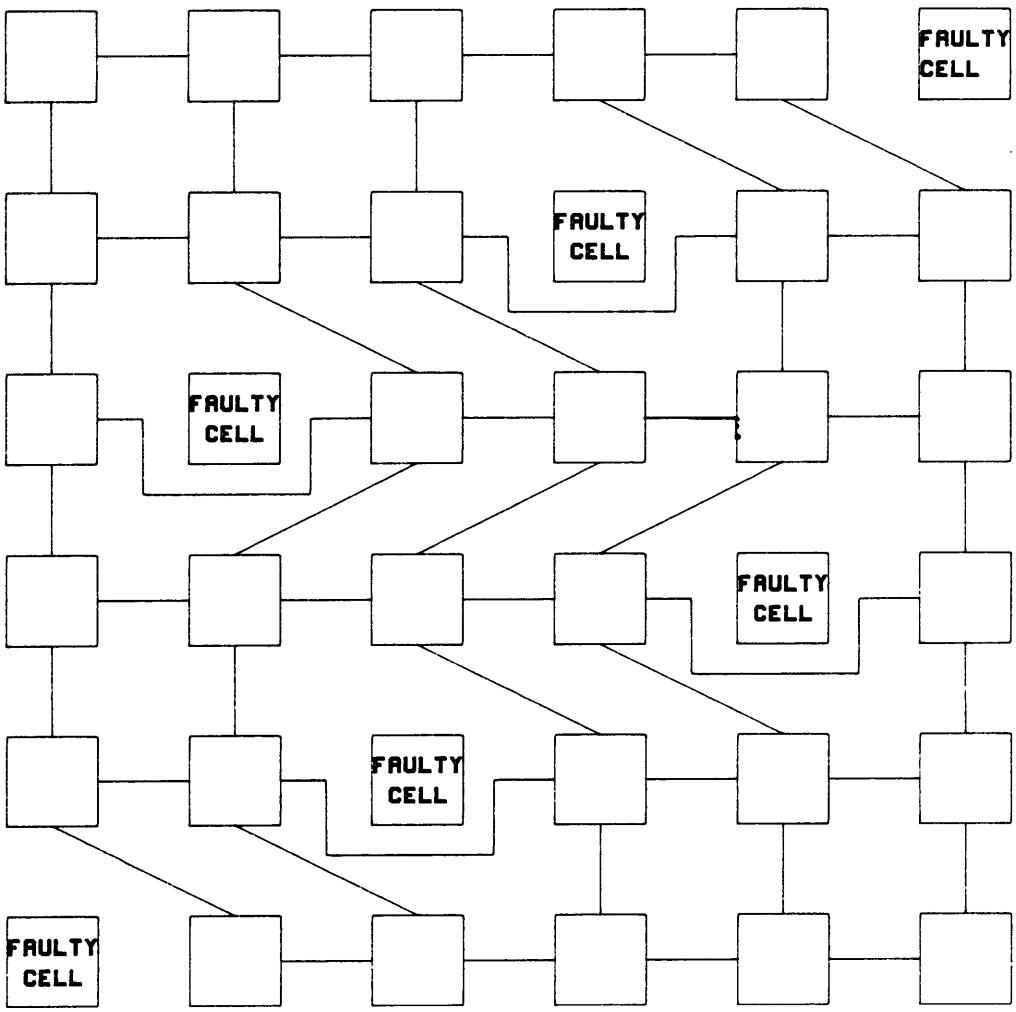


Figure 10. Multiple Single Fault Occurrences: Array Showing The Effects of a Single Faulty Cell in Each Row.

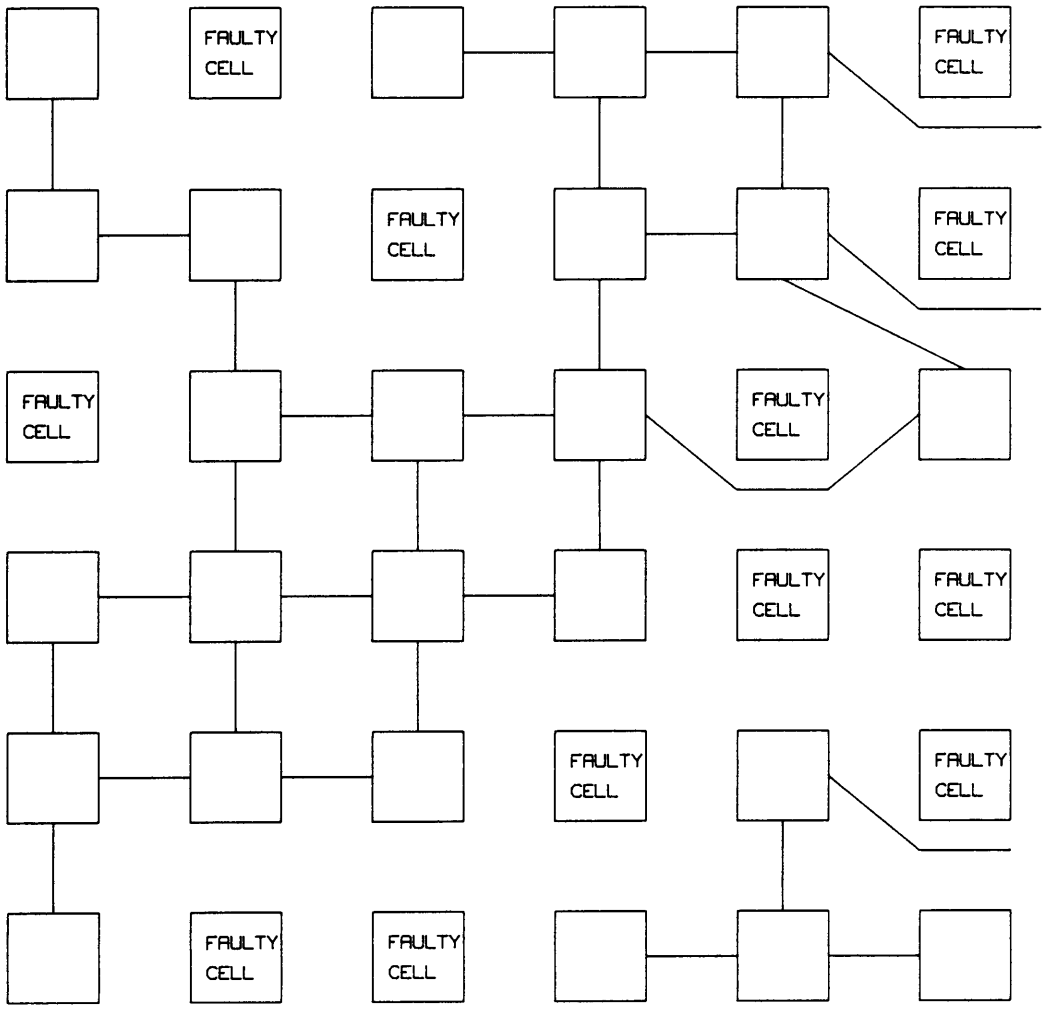


Figure 11. Double Fault Conditions: Array Showing Maximum Effect of Two Faulty Cells in Each Row.

neighbors. Cells in the same row as the multiple faults lose either a west or east neighbor, and cells to the south of faulty cells have no northern neighbor. These cells behave identically to cells on the array exterior. Multiple faulty cells in a row reduce the number of full length columns available on a one- to-one basis.(Figure 12 on page 36)

The occurrence of single faults in rows to the north or south of a row with multiple faults is handled much the same as normal single faulted rows. If the single fault is to the west of a multiple fault in a southern row, all cells in both rows take on the same configuration that would have existed had there not been a multiple fault. All fault-free cells in the northern row shift communications to the southern row by one cell to the west, up to the cell immediately northeast of the last faulty cell(one). Cells to the east of cell(one) enable connections to their cell to the south (Figure 13 on page 37). This leaves the cell to the north of the second faulty cell connected to the cell west of the faulty cell, assuming neither of these cells is faulty. The cell that would normally be connected to the faulty cell(two) now must disable its communications to the south. This column now becomes the segmented, or a column in two parts. If the columns were numbered at the cell boundaries however, the segmented column will remain the same number as before because of reconfiguration that has taken place to the north (Figure 14 on page 38).

The existence of a single fault in a row south of a multiple fault row is treated much the same as the previous example.

2.1.2.1 SPECIAL CASES

When multiple faults exist in rows, they create some special problems with communications of the fault conditions in portions of the array to the east and west of the multiple faults.

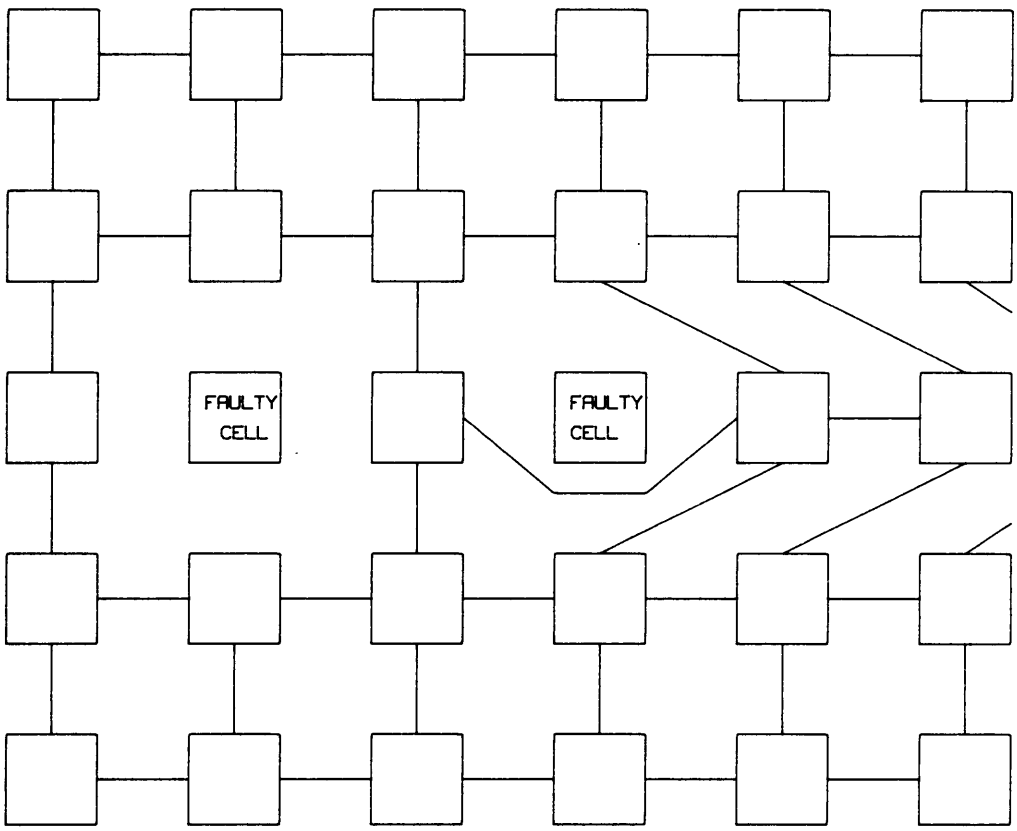


Figure 12. Multiple Faults: Array Showing the Effects of a Second Fault.

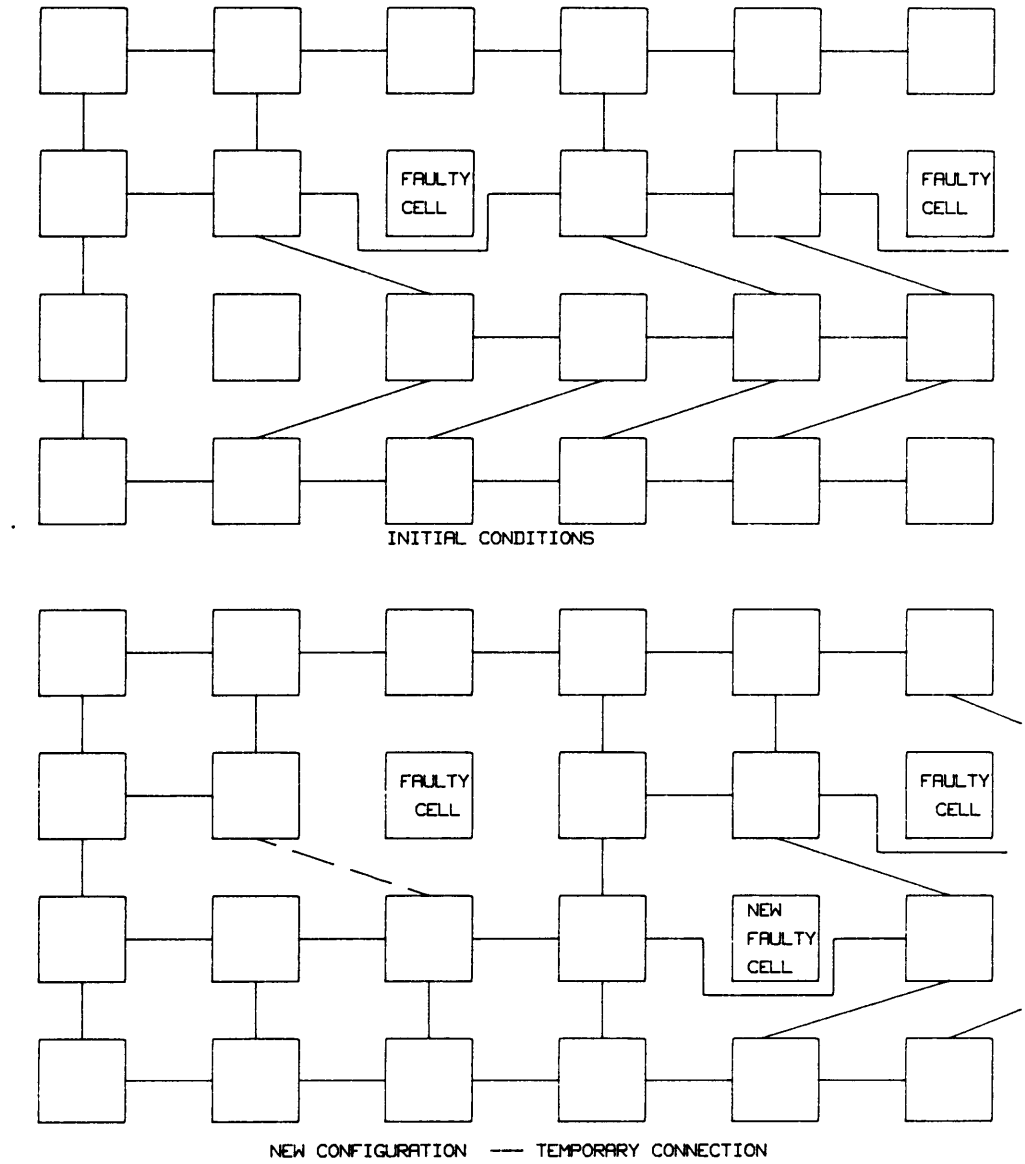


Figure 13. Double-Fault with Single Fault: Double Fault Bounded by a Single Fault to the North.

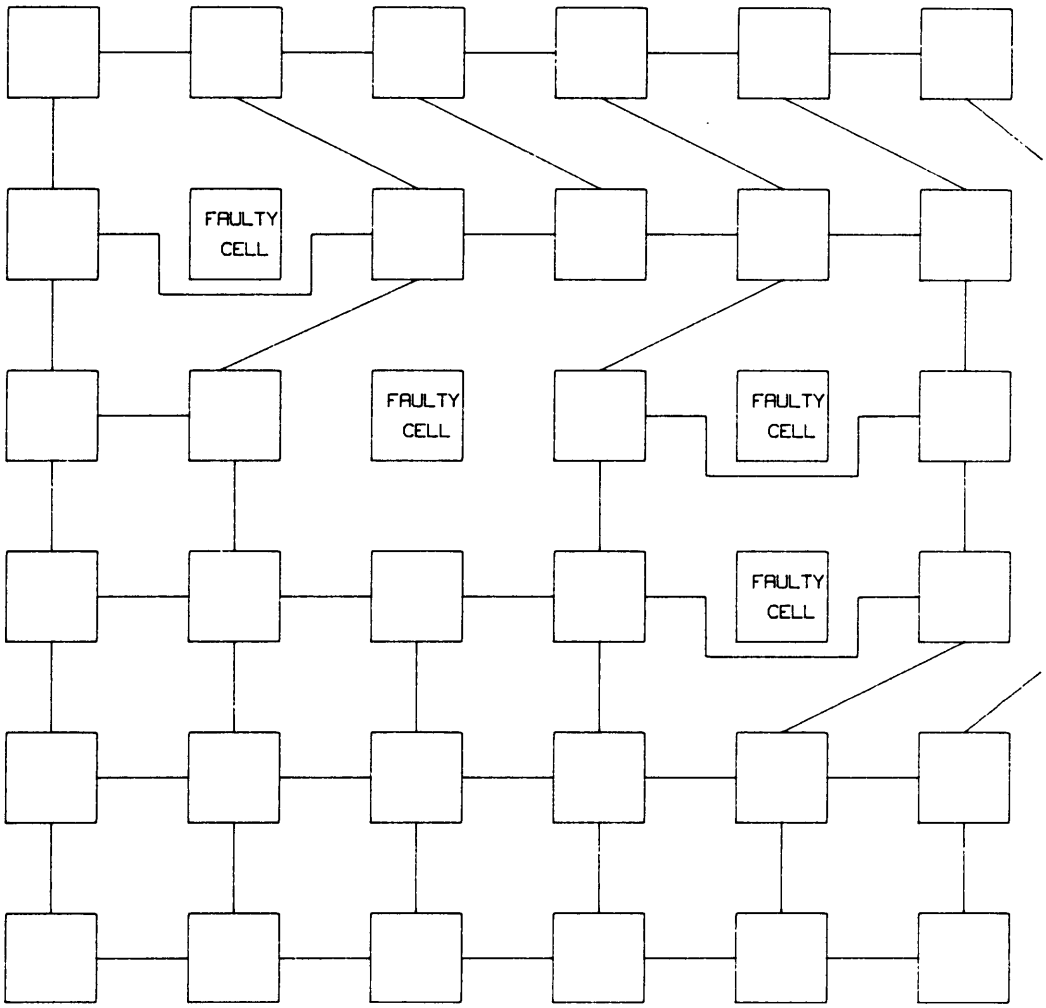


Figure 14. Array With Single and Multiple Faults: Showing The Effect Of Single Faults to The North or South of A Multiple Fault.

Normally, when a cell detects a fault, it immediately takes on the proper configuration to route around the fault. This causes problems when multiple faults occur.

THE SECOND FAULTY CELL: If a row with a single faulty cell suffers the occurrence of a second faulty cell to the east of the first fault, information about the location of the second fault is communicated along the row until it reaches the cell to the immediate east of the first faulty cell. Normally, this cell would immediately disable its communications to the west. This would leave the cell to the west of the faulty cell without the knowledge of the multiple fault. This cell would still attempt to communicate with the cell to its far-east. This causes no immediate problems since the communications channel is broken by the other cell. However, it does create problems later should faults occur to the northwest or southwest. These cells have the information that a multiple fault exists in the northern or southern row, while the western portion of that row does not have this information. For configuration to proceed correctly, both rows must have this information.

The above problem, and several other similar problems, are corrected by forcing cells to open communications channels they would normally close. In the above case, the cell to the east of the multiple faulty cell(two), is forced to maintain row communications for one clock cycle past its normal reconfiguration time. This allows it to communicate the existence of the eastern faulty cell to the western section of the array row.

A similar problem exists for a faulty cell occurrence to the northwest or southwest of a multiple fault. Assume row A has a multiple fault condition, and a fault occurs in row B, which can be either the northern or southern row. In this text, row B will be the southern row, but the discussion applies equally to the northern row. When the fault occurs in row B, to the west of the multiple fault in row A, information about the fault condition is communicated in row A until it reaches the multiple fault. There is no row communications to the east from this cell. The information about the fault must be picked up by row A from cells in row B to the east. However, cells in row B will immediately change their connections to row A as soon as they learn

of the new fault condition. Since row A does not know of the condition, and row B does, all communications between the two rows will be broken (Figure 15 on page 41).

This problem is eliminated by forcing the cell in row B to the south of the next fault-free cell in row A to maintain communications due north for one extra clock cycle. This allows the information about the fault in row B to be communicated to row A. If there are more than two faulty cells in row A, this process will be repeated as many times as is necessary to insure all of row A has the information.

Identical circumstances exist if row B contains a faulty cell to the immediate northwest or southeast of the multiple fault in row A (Figure 17 on page 43). Should a second faulty cell occur in row B to the east of the multiple fault, row A will not be able to communicate that information past the isolated multiple fault. Again the information must be communicated from row B. However, this time the communications must come from the cell in row B either to the north or south of the multiple fault. A method similar to the above is used to force this cell to maintain communications with row A for one additional clock cycle.

These delays are forced to occur by adding an additional eight-bit register, which is used to indicate the location of a multiple fault relative to a cells location within the array. These bits indicate the presence of a multiple to the immediate west, east, north, south, northeast, northwest, southeast or southwest. The logic controlling the north, south , east and west bits can come directly from the fault register. For example, the multiple-fault-west bit would be set if a cells fault register indicated that the cell to the west was faulty, and a cell somewhere to the east was faulty. Double-fault-south would be set if a faulty cell existed to the south and another was to the south-east. These four bits could be added to those transmitted to other cells. Since the remaining conditions, northeast, northwest, southeast and southwest might require another reconfiguration and testing cycle before they were detected, those four bits would be set when the input from the other cells dictated. For instance, the double-fault-

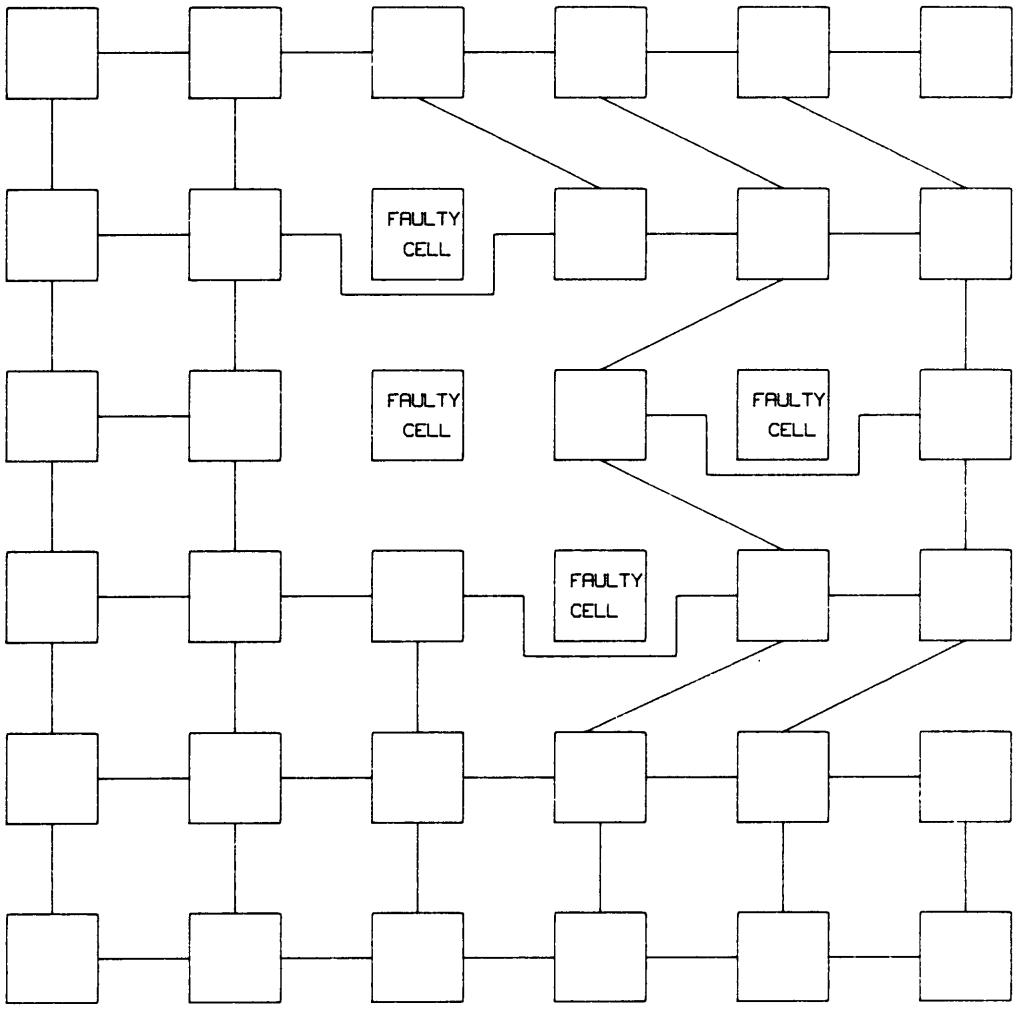


Figure 15. Multiple Fault Bounded By A Single Fault

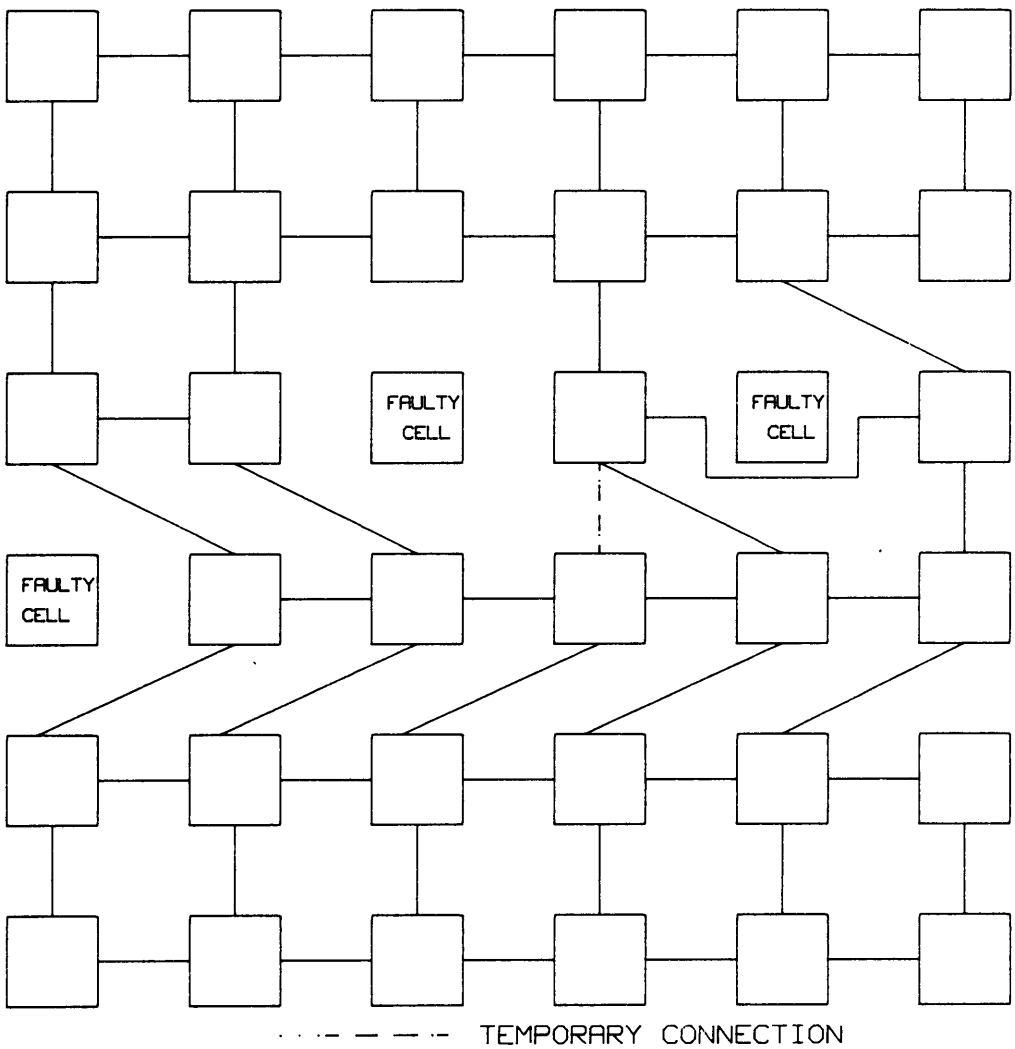


Figure 16. Multiple Fault Bounded By A Single Fault: Showing Intermediate Interconnect Scheme

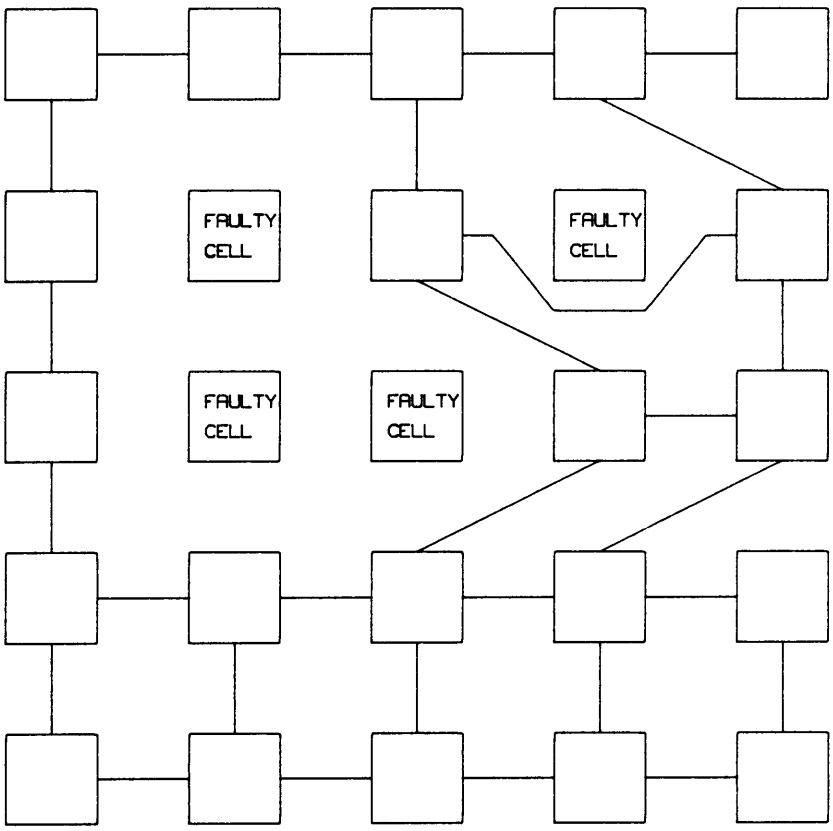


Figure 17. Double Fault Bordered by a Double Fault

northeast would be set if either the input from the western cell indicated a double-fault-north, or when the input from the north indicated a double-fault-west.

The contents of this register, together with the changing contents of the fault-register are sufficient to trigger the circuits controlling the communications channels to enable certain paths for an additional clock cycle.

2.2 PROOF OF THE ALGORITHM

This algorithm correctly reconfigures in the presence of any configuration that involves only a single faulty cell in any row of the array.

Proof: by induction on the number of rows.

If a single row suffers the occurrence of a faulty cell, cells to the east of the faulty cell, in the same row, shift communications with rows to the north and south one cell to the west. The cells in the north and south rows, north or south or east of the faulty cell, shift communications with the row containing the faulty cell one cell to the east. The cells on the eastern border of the array in these two rows loose their neighbor in the row with the faulty cell.

If faults occur in two adjacent rows, the communications of the row to the north of the faults is as it was in the above case. The row to the south of both faults also behaves as the above case indicates. Only the communications configuration of the two rows involved changes. Cells to the west of both faults are not affected. The communications of cells to the east of both faults reverts to the straight north/south scheme, as if there were no faults. If the fault in the northern row is east of the fault in the southern row, then communications of cells between the faults is shifted to the east in the northern row, and the west in the southern row. If the fault in the northern row is west of the southern fault, communications of the cells be-

tween the faults is reversed, with the southern cells communicating to the northeast, and the cells in the northern row communicating to the southwest. From this case, it is clear that only faults in two rows determine the communications configuration between the two rows.

The configuration algorithm works for cases of only one fault, or faults in two adjacent rows. Assume it works for arrays of N rows. Show that it works for N+1 rows.

Since we are only proving the cases where single faults in a row are tolerated, if we have N+1 rows, we can have at most N+1 faulty cells. If we limit our discussion to the faulty cell in row N+1, the communications between row N and row N+1 is determined only by the faults in these rows. We have already shown that the algorithm will configure around single faulty cells in two consecutive rows, so the communications paths between these two rows are valid. Therefore, communications is valid for any number of single fault occurrences in an N+1 row array.

3.0 APPLICATION TO THE CELLULAR ARCHITECTURE

Effective application of the reconfiguration algorithm to the cellular array [92] will require a small amount of modification to the existing array and reconfiguration algorithms. The current six cycles, consisting of transmitting s-value (one and two), calculation of new s-value (three), transmitting current-state (four and five), and calculation of the new state (six) must be modified slightly to allow transmission of the fault- location data and the fault isolation data. In the following text, the fault-isolation data, which is the four bits defining the location of a multiple fault, is assumed to be transmitted with the s-value data. The ten bits of the fault-register, together with the other signals that must be added for this particular application, are transmitted with the current- state data. This scheme is by no means a requirement, and any real application should divide the data to make the most economical use of the time available.

The original cellular architecture [92] has the clock cycle divided into six periods, or twelve clock transitions, either negative or positive, depending on the logical implementation. Diagnostic testing and data transmission are done on clock transitions of opposite polarity. A cell might have the following sequence of events, depending on the cell priority.

1. Transmit S-value.

2. Test.

3. Receive S-value.

4. Test.

5. Calculate S-value.

6. Test.

7. Transmit Current-State.

8. Test.

9. Receive Current-state.

10. Test.

11. Calculate New Current-state.

12. Test.

Although it is not required that a particular cell be tested at the same time by all cells with which it communicates, it is important that the cell be tested completely within the same major cycle. For the reconfiguration algorithm to work as discussed, all four cells testing any one cell must determine whether it is faulty or not within the same major cycle, or within the same six clock periods. This is necessary to insure reconfiguration as quickly as possible, and to insure all output data from the execution plane is valid data. The preceding sequence is given

only to list the number of possible timings for test sequences, and in not intended to indicate the number of required sequences.

Additional modifications could include logic to eliminate the propagation of faults from outside the active portion of the array. Any fault that occurs outside the pattern of the working array, either to the left or the right, need not be communicated to the active cells, or the spare column of cells. This is very easily accomplished by adding a simple check to the incoming data. If we assume cells in the quiescent state have current states of zero, and spare cells have current states of one, then because each cell knows the state of all cells with which it communicates, if a cell has a non-zero state, and one of its neighbors has a current state of zero, its data concerning the locations of faults can be ignored. The data is transmitted each clock cycle, and when the array clears itself to prepare for the generation of another pattern, or for the regeneration of the same pattern after a multiple fault occurrence, the fault data can be transmitted to those cells within the old pattern. Addition of this logic will allow the active array to respond only to faulty cells within the pattern, much the same as it did before inclusion of the reconfiguration logic.

Logic already exists within the cellular array to notify the outside world, the device that is the source of data to be processed, that a fault has occurred. This logic must be modified to interrupt the flow of data while either the pattern is being regrown for a multiple fault in the same row, or for a reconfiguration after a single row fault. This means the addition of another signal to the output of the array, but not necessarily the addition of another external connection. Both of these signals could be generated in the form of coded data to indicate the internal status of the array. These signals could be passed to the external device at times when that device is not expecting computed data. Since all cells, both in the control and the execution hyper-plane, have testing cycles, the status data could be transmitted by a cell with external connections during the testing cycle. These signals would interrupt the flow of data during the reconfiguration, and signal the external device to restart data-flow after completion of the reconfiguration.

3.1 BEHAVIOR AFTER FAULT OCCURRENCES

Addition of the reconfiguration logic to the cellular array will enable greater amounts of data to be processed by an array that suffers the occurrence of a faulty cell.

3.1.1 THE EXISTING CELLULAR ARRAY

Suppose a cellular array already exists with the following state configuration.

0 0 0 0 0 0 0 0 0 0

0 A B C 0 0 0 0 0 0

0 D E F 0 0 0 0 0 0

0 G H J 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

Cells with states of A, B, C, D, E, F, G, H, and J form the active portion of the array. These cells perform the computations to execute the algorithm. Cells with states of 0 are quiescent cells.

Suppose a fault occurs in the cell with the current state of E. The array clearing and neutralization algorithms will return the states of all cells to 0, the quiescent state in our example, with the exception of one cell, which acts as the seed cell for pattern regrowth. The

cycle might proceed in this manner.

```
0 0 0 0 0    0 0 0 0 0    0 0 0 0 0    0 0 0 0 0    0 0 0 0 0
0 A B C 0    0 A B C 0    0 A 0 C 0    0 0 0 0 0    0 0 q 0 0
0 D E F 0    0 D X F 0    0 0 X 0 0    0 0 X 0 0    0 q X q 0
0 G H J 0    0 G H J 0    0 G 0 J 0    0 0 0 0 0    0 0 q 0 0
0 0 0 0 0    0 0 0 0 0    0 0 0 0 0    0 0 0 0 0    0 0 0 0 0
```

The neutralization and clearing cycles take a large number of clock cycles, depending upon the overall size of the array. At the end of this cycle, one of the isolation cells will still contain the global state. It will eject this seed into the fault-free area of the array, and assuming no other faults exist, the pattern growth process will take three clock cycles to locate sufficient fault-free area, and three cycles to regrow the pattern and reach the steady state condition again.

The number of major cycles required to detect the faulty cell and regenerate the pattern in a fault free section of the array is seven plus the array neutralization time, in this example. The amount of time required might be much greater if faulty cells exist to the right of the active pattern. The search by the seed cell for a contiguous fault-free area may be complicated in this case and require much more time. This does not include the time required to allow the entire computational plane to reconfigure, or form the communications paths necessary to restart computation.

3.1.1.1 FAULTY CELL ISOLATION

From the above example, is should be noted that the new pattern was regrown in an area that left one fault-free cell between the active pattern and the faulty cell. This is accomplished by setting the s-values of cells in communication with the faulty cell to a "-1". Looking at the new

s-value distribution, we have

0	0	0	0	0	0	0	0	0
0	0	-1	0	1	1	1	1	0
0	-1	X	-1	0	1	2	1	0
0	0	-1	0	1	2	2	1	0
0	1	0	1	2	3	2	1	0
0	1	1	2	3	3	2	1	0
0	1	2	2	2	2	2	1	0
0	1	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0	0.

The value of s for any cell can be no greater than one more than the s-value of any of its neighbors. This forces all cells adjacent to the isolation cells to take on an s_value of zero. Other s_values will increase from the boundary until they reach steady state values. The 3 by 3 pattern will require an s-value of "2" to begin growth, so the pattern will be regrown four cells to the right of the faulty cell. The seed migration algorithm will transmit the seed from cell to cell until it locates a cell with an s_value of 2. One fault free cell is left between the new pattern and the faulty cell because of the isolation algorithm, and an additional cell is left because of the inefficiencies of growing a square pattern from a diamond growth pattern.

3.1.2 AN ARRAY WITH THE NEW RECONFIGURATION ALGORITHM

Now suppose the same array has the new reconfiguration algorithm as a part of its logic. The

original pattern would be this. 1s

0 0 0 0 0 0 0 0 0 0

0 A B C 1 0 0 0 0 0

0 D E F 1 0 0 0 0 0

0 G H J 1 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

Cells with a current state of 1 have been added to the end of each row. These form the spares that allow the reconfiguration to take place. These cells perform no computations during normal operation. The action of the array cells with reconfiguration is to detect the single fault, and shift operations assigned to each cell to a cell to the right, with the spare cells now becoming part of the active pattern. This would proceed as follows.

0 0 0 0 0 0 0

0 A B C 1 0 0

0 D X F 1 0 0

0 G H J 1 0 0

0 0 0 0 0 0 0

0 0 0 0 0 0 0

0 A B C 1 0 0

0 D X E 1 0 0

0 G H J 1 0 0

0 0 0 0 0 0 0

0 0 0 0 0 0 0

0 A B C 1 0 0

0 D X E F 0 0

0 G H J 1 0 0

0 0 0 0 0 0 0

This reconfiguration operation required only two clock cycles after the fault occurrence, compared with the large number under the existing algorithm. With the existing algorithm, the number of clock cycles required to clear and neutralize the array is a function of the a The original algorithm would have required all nine cells in the computational plane form new communications paths. With reconfiguration, only two cells must perform this operation. In this case, faulty cell isolation can be accomplished by disabling communications networks, so the isolation cells are not required. There is a penalty for transmitting the fault-status bits required to accomplish the reconfiguration, but if the number of bits now transmitted each clock cycle is less than the number of data bits transmitted in the execution plane, there may

not be a time penalty. This assumes the computational and control plane cells use the same clocking mechanism.

The new s-value distribution for this same array with reconfiguration capabilities is

0 0 0 0 0 0 0 0 0

0 1 1 1 1 1 1 1 0

0 1 X 2 2 2 2 1 0

0 1 2 3 3 3 2 1 0

0 1 2 3 4 3 2 1 0

0 1 2 3 3 3 2 1 0

0 1 2 2 2 2 2 1 0

0 1 1 1 1 1 1 1 0

0 0 0 0 0 0 0 0 0.

Comparrison of this distribution with that on page 51 will show the advantages of network reconfiguration.

3.2 PATTERN GROWTH AROUND FAULTY CELLS

The cellular array is designed to handle operations on more than one algorithm if necessary. To do this, ways must be found to change the existing pattern, either by insertion of seed information in a cell by outside communications, or through internal programming within the cell.

3.2.1 EXTERNAL SEED INJECTION IN THE CURRENT ARRAY ARCHITECTURE

If the array has completed operations on an algorithm, for example one which required a three by three matrix, and wishes to execute an algorithm which requires locating a cell with an s-value of three, the existence of faulty cells that were outside the original matrix can create problems. If outside seeding is to be done, it is reasonable to assume the seed injection will be done with a cell that has established communications with the external device. This could be a cell that was a part of the old pattern.

If the old pattern required a three by three area, it would have required an s-value of 2 in order to grow its pattern. For this discussion, we will assume the array has completed the clearing operation, and is ready to accept a seed. A cell which was not within the original active part of the array has become faulty. The state structure of the array might be

0 0 0 0 0 0 0 0 0 0

0 S 0 0 0 0 0 0 0 0

0 0 0 0 X 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

where s is the seed cell, and x is the faulty cell. The new seed has been deposited in the top-left-corner of the original three by three. Since the faulty cell did not occur within the previous active region, no action would have been required to complete execution of that task. The s-values are now recomputed before growth of the new pattern begins. The s-

values before occurrence of the fault would have been

0 0 0 0 0 0 0 0 0 0

0 1 1 1 1 1 1 1 1 0

0 1 2 2 2 2 2 2 1 0

0 1 2 3 3 3 3 2 1 0.

(The bottom portion of the array has been eliminated for this example.) With the faulty cell, the values of the s-value will change to

0 0 0 0 0 0 0 0 0 0

0 1 1 0 -1 0 1 1 1 0

0 1 0 -1 X -1 0 1 1 0

0 1 1 0 -1 0 1 2 1 0

0 1 2 1 0 1 2 2 1 0

0 1 2 2 1 2 3 2 1 0.

Those internal cells surrounding the faulty cell (x) with an s-value of -1 are now isolation cells. It is obvious that the seed, which is in the upper-left cell with an s-value of 1, must travel much farther than in the fault- free array to find the cell with an s-value of 3. In the fault-free cell, the seed would require only four clock cycles to reach the desired cell. With the fault, the seed must have six cycles before reaching the desired cell. In both cases it was assumed the seed would travel downward until it reached a cell with a higher s-value in a cell to the right. It would then move to the right of that cell.

3.2.1.1 REGROWTH WITH THE NEW RECONFIGURATION ALGORITHM INSTALLED

If the reconfiguration logic is now added to the cellular array, the calculation of new s-values after the fault occurrence will be changed greatly. With reconfiguration, the calculation of s-values will yield the array

0 0 0 0 0 0 0 0 0 0

0 1 1 1 1 1 1 1 1 0

0 1 2 2 X 2 2 2 1 0

0 1 2 3 3 3 3 2 1 0.

If the seed is deposited in the same upper-left corner of the array, the time required to locate a cell with an s-value of 3 will remain at four clock cycles, the same time required for a fault-free array. This capability is created because the faulty cell is now isolated with switch circuitry instead of cells with s-values of -1. Therefore the s-values of all cells in the vicinity of the faulty cell are increased.

3.3 OCCURRENCE OF A FAULT DURING PATTERN GROWTH

For discussion, assume we are generating a three by three pattern, which requires an s-value of 2. A faulty cell is detected before the pattern growth is complete. The cell pattern and s-

values at the time of the fault are

STATE	S_VALUE
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 a 0 0 0 0 0	0 1 1 1 1 1 1 0
0 b c d 0 0 0 0	0 1 2 2 2 2 1 0
0 0 e 0 0 0 0 0	0 1 2 3 3 2 1 0
0 0 0 0 0 0 0 0	0 1 2 2 2 2 1 0
0 0 0 0 0 0 0 0	0 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0.

A fault occurs in the cell with a current state of d, and an s-value of 2.

3.3.1 WITHOUT RECONFIGURATION

If the isolation algorithm produces a seed in the cell which has a current state of c, the configuration of the array after clearing will be

0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 1 0 -1 0 1 1 0
0 0 s X 0 0 0 0	0 0 -1 X -1 1 1 0
0 0 0 0 0 0 0 0	0 1 0 -1 0 1 1 0
0 0 0 0 0 0 0 0	0 1 1 0 1 2 1 0
0 0 0 0 0 0 0 0	0 1 2 1 2 3 1 0

The seed must travel three cells downward to reach a cell with an s-value of two.

3.3.2 PATTERN REGROWTH WITH THE NEW RECONFIGURATION ALGORITHM

If the same fault occurs at the identical time in an array which has reconfiguration capabilities, the s-value configuration after detection of the fault will be

0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 0
0 1 2 X 2 2 1 0
0 1 2 3 1 2 1 0
0 1 2 2 2 2 1 0
0 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0.

Notice the cell in the center of the array with an s-value of 1, is surrounded by cells with a minimum s-value of 2. This is the result of the wave-effect of the reconfiguration, which temporarily leaves this cell with only three interconnections. In the reconfiguration algorithm, this cell reads the absent s-value input as 0, and calculates its new s-value to be 1. It will take several clock cycles to propagate this change throughout the array. The sequence of s-values will be

0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 0	0 1 1 1 1 1 1 0	0 1 1 1 1 1 1 0
0 1 2 X 2 2 1 0	0 1 2 X 2 2 1 0	0 1 2 X 2 2 1 0
0 1 2 2 3 1 1 0	0 1 2 3 2 2 1 0	0 1 2 3 3 2 1 0
0 1 2 2 2 2 1 0	0 1 2 2 2 2 1 0	0 1 2 2 2 2 1 0
0 1 1 1 1 1 1 0	0 1 1 1 1 1 1 0	0 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0.

Depending on the size of the array, the time for the s-values to reach their final state will vary. For this reason, and because the clearing and neutralization algorithms must be

executed,even though t be regrown in the same general area of the array, there will be no advantage over the non-reconfigurable array. In both cases, time must be allotted to allow the new s-values to be generated before pattern regrowth can be restarted. This time is used by both the arrays to execute the isolation algorithm to determine the location of the new seed cell. The only advantage of the reconfigurable array is the more efficient use of fault-free cells, and the ability to restart the pattern growth process without seed migration, since the seed will be created after isolation in a cell with an s-value of 2.

3.4 MULTIPLE FAULTS

When multiple faults occur, decisions must be made concerning the generation of a new seed cell. The reconfiguration algorithm can only handle a single fault before requiring regeneration of the cell pattern in some other section of the array. Two possibilities exist, either the eastern faulty cell must trigger the seed generation, or the western fault must do so.

3.4.1 TWO FAULTY CELLS

The occurrence of multiple faults is treated much as single faults without the reconfiguration algorithm applied. Two possibilities exist for the sequence of multiple faults, either both cells become faulty at the same time, or they fail in sequence. If we decide to isolate the farthest west, the following sequence will take place.

Should two cells fail, one at a time, in the same row of the active pattern in the array, a determination is made about which cell is the farthest west by testing the contents of the fault-register. Only one cell will require isolation. This cell will be the one farthest west, and will

be isolated by the action of the reconfiguration algorithm. The cells surrounding this isolated faulty cell are used to regenerate a seed cell. The existing array clearing and neutralization algorithms can now to generate a new seed cell, which begins the pattern regrowth process.

After the initial cell failure, the array will configure around the faulty cell. When the second cell failure occurs, if it is west of the initial failure, the seed cell generation algorithm can be started immediately. Assuming the initial reconfiguration has completed and the array is in execution mode again, all cells in the rows adjacent to the initial failure will know of the failure. If the second fault occurs west of the first, the cells surrounding the second fault will know that a multiple fault exists, and proceed with seed generation.

If the second failure occurs to the east of the initial fault, the cells surrounding the second failure will know of the multiple fault, and will notify the external devices. The array clearing and neutralization process cannot begin until information about the second fault has propagated to the cells surrounding the initial fault. At this time, the seed generation process can begin. This will happen at a time later than if the two cells had failed in opposite order. The amount of time required to recover from a multiple fault depends upon the sequence of the failures.

3.4.2 SIMULTANEOUS FAULTS

Simultaneous fault occurrences will initially start reconfiguration as if only a single fault had occurred. Recognition of the double fault condition will not be realized until propagation of all fault data is completed. At this time, the mode of operation will be changed to that of seed regeneration, which will again generate the seed cell in the vicinity of the western-most fault.

3.5 CELL PRIORITY

Cells are given a transmission priority because of the multiplexed communications scheme. During reconfiguration, cells with the same priority will temporarily be connected with cells of the same priority. Cells only communicate with cells of the opposite priority, so when reconfiguration is attempted, the cells to the north, south, and east of the faulty cell will attempt to enable a communications network that connects the three cells in a north-south line. However, since all three cells have identical priority codes, no communications will be possible.

A change in the priority of the cells must be accomplished. The cells in the north and south rows must continue to communicate with three of the four previous cells in their neighborhood, so it is not reasonable to change the priority of these cells, as this leads to a change in the priority of nearly every cell in the array. The cell to the east of the faulty cell must continue to communicate with the cell to its east, so it must hold the same priority as before at least until the faulty information is passed to the eastern cell. This can be accomplished by using the same "reconfigure" signal used to cause the cells to take on the state configuration of their western neighbors during reconfiguration. After this signal and the fault information has been communicated to the eastern cell, the priority of the cell will be changed. This process will be followed down the entire length of the array row, including cells in the quiescent state. Although these cells are outside the active pattern, their priority must be altered, together with their interconnect scheme to maintain the proper communications with neighboring cells.

3.6 HARDWARE IMPLEMENTATION

Hardware implementation of the reconfiguration algorithm requires that some logic be added to the existing cell architecture. To control which four of the ten communications lines are enabled, transistor switches are added to each link (Figure 18 on page 63.). This means a link between two cells must pass through two of these switches, and therefore both cells must be attempting to enable the link before communications is established. This provides the method of isolating faulty cells, since the faulty cell may be trying to establish a link with all cells in its neighborhood. The switch logic is vital to fault-free communications, so diagnostics must be able to determine not only the locations of faulty cells, but also faulty switches. If this is not available, the communications network must be designed to be fault-tolerant.

The control signals for the switches are derived by decoding the ten bits of the fault register. To accommodate these extra bits, the input register for each of the input lines must be increased by twelve bits. This number could be decreased by sharing the transmission of the fault register data between the state and s-value cycles.

Logic for decoding the fault register could consist of a PLA. The input to the register could be implemented as shown in Figure 19 on page 64 and Figure 20 on page 65. The logic is relatively simple, and if the existing architecture generates a large cell size, the reconfiguration logic should add little to the complexity of the circuitry.

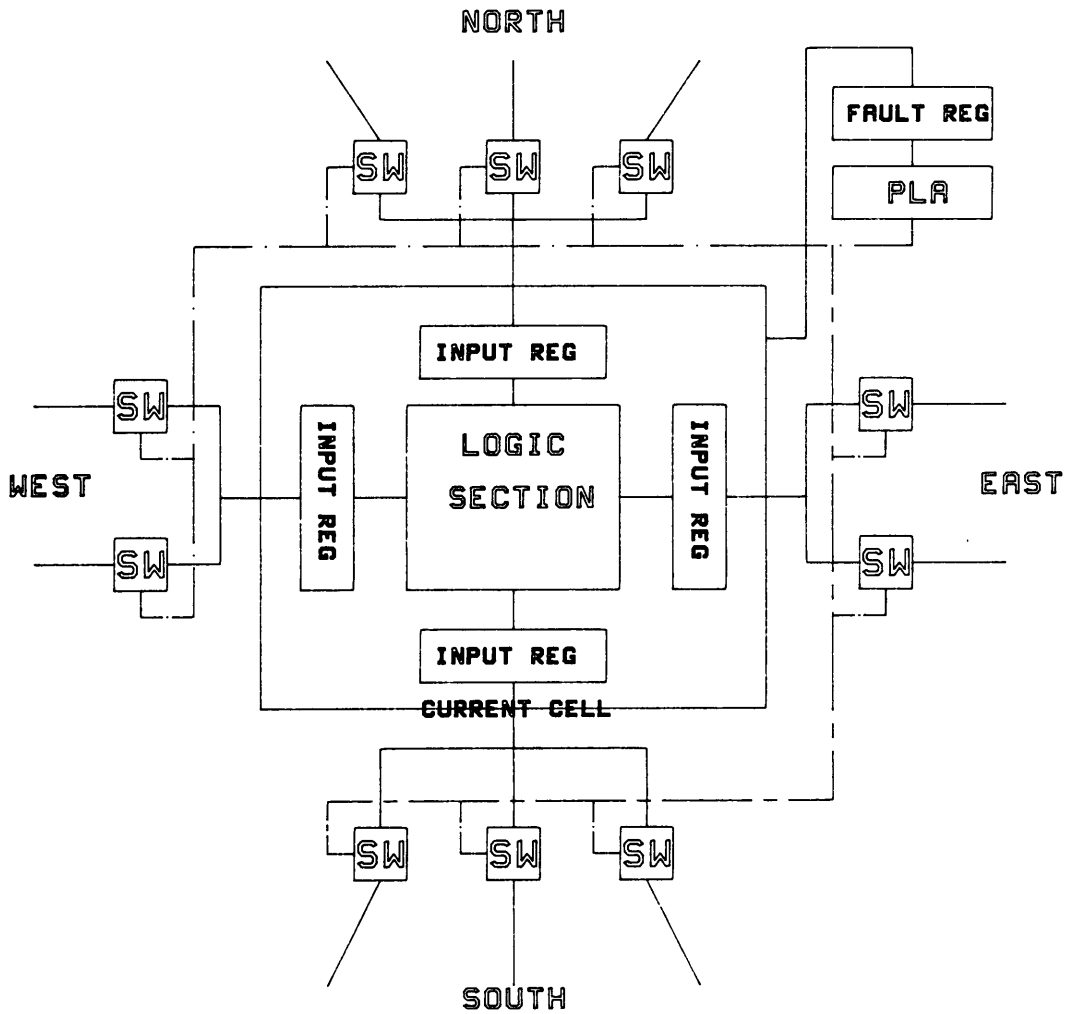


Figure 18. New Cell Block Diagram: Showing Additional Communications Links, Switches and Decode Logic

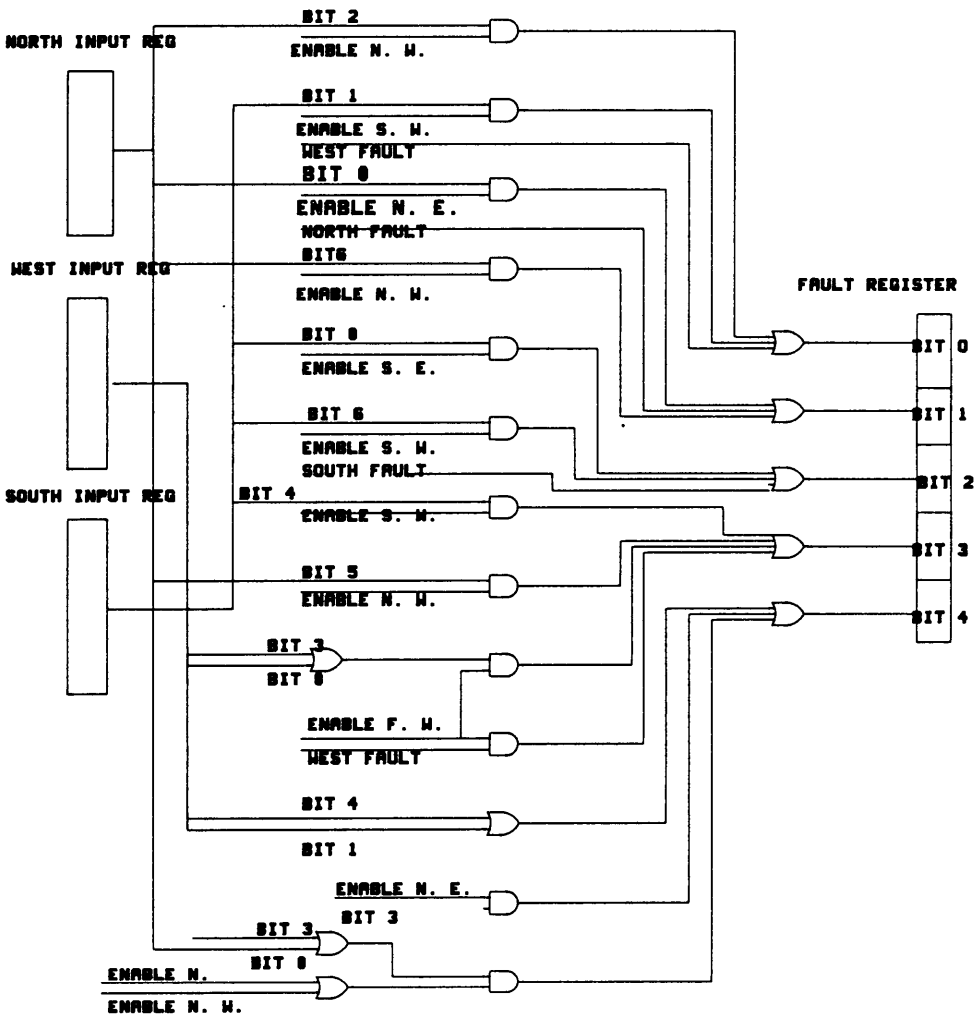


Figure 19. Fault Register Logic: Bits Zero Through Four (and-or Implementation)

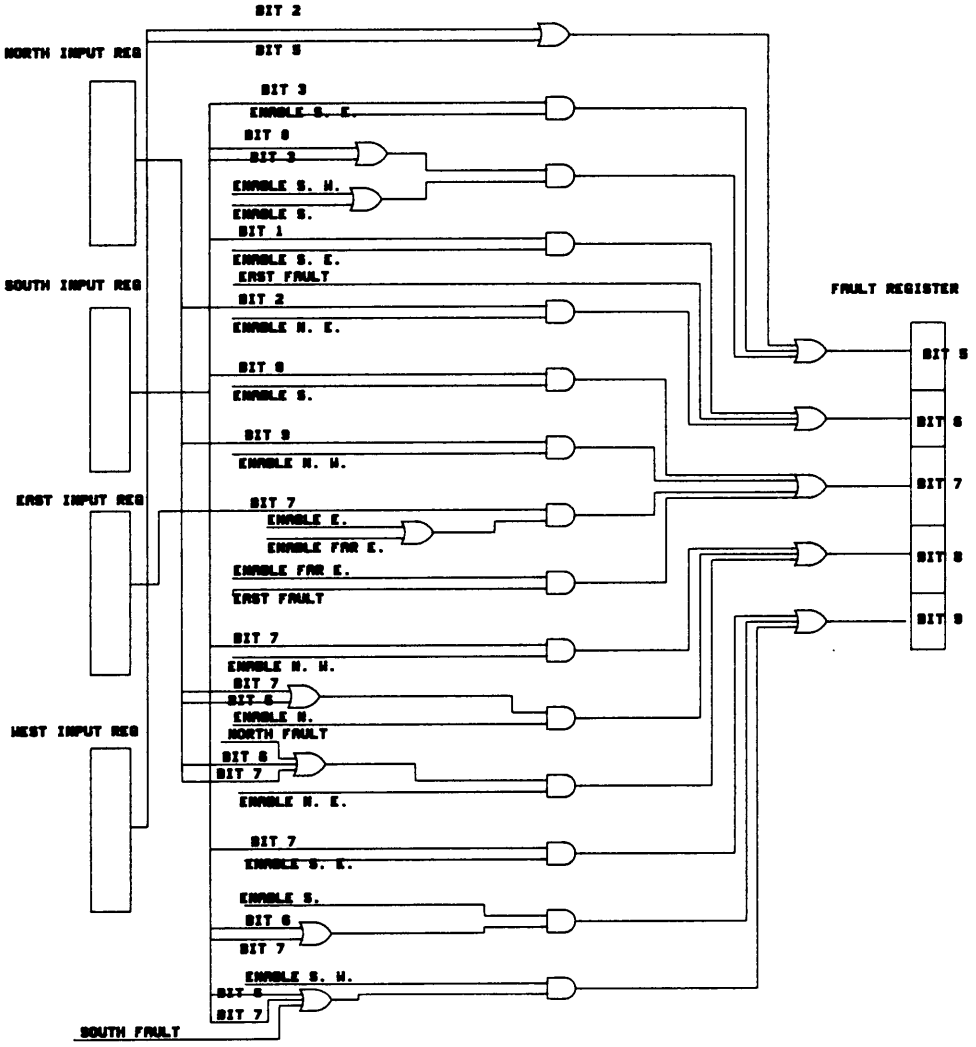


Figure 20. Fault Register Logic: Bits Five Through Nine (and-or Implementation)

4.0 ADDITIONAL ENHANCEMENTS

While the addition of the reconfiguration algorithm to the cellular architecture greatly enhances the fault tolerant characteristics of the array, some deficiencies in the array algorithm still require further work. One of these is the current s-value algorithm [92] and its inefficiencies.

4.1 S-VALUE REQUIREMENTS

The current s-value algorithm assumes the only information required is the maximum size of a diamond pattern that can be generated in the neighborhood of a cell. However, using the diamond shape to grow pattern of other regular configurations, such as squares or rectangles, is very inefficient. If we wish to grow a three by three square matrix in the array, we must begin with a five by five diamond. This means that of the thirteen cells required to grow the diamond, only seventy percent of the cells will remain in the final pattern. The remaining four cells on the corners of the diamond will have a final next-state transition to the quiescent state.

With larger square configurations, the inefficiencies become even more pronounced. A five by five or a four by four square will require a nine by nine diamond. The efficiency of the five by five is approximately 60 percent, with the four by four being much worse. In addition to these inherent inefficiencies, if the overall shape of the array is a square or rectangle, no more than four cells in the outside perimeter can ever by used for a pattern, those serving as the corners of the diamond. If we do not configure for a diamond or triangle shaped algorithm, cells in the outside perimeter will never be used, and therefore serve only to aid in growing patterns, but never in execution of an algorithm. Unless the size of the array can be made very large, while the executable algorithms remain very small, a large percentage of the cells in the control plane will be used only occasionally, while those cells in the execution plane would never by accessed. It might be possible to generate the array layout with only control cells in the outside perimeter to conserve silicon resources, but this would partially defeat the advantages of the parallel processor array by forcing the manufacturer to use more masks for array layout.

If a way can be found to grow the square pattern directly, without first growing the much larger diamond, cell usage efficiency can be greatly increased. To accomplish this, ways must be found to calculate the s-value in such a way that it indicates the maximum size of a square pattern in the cell neighborhood. The possibility of keeping both s-values might be considered, since the memory requirement is small in the case of the present s-value algorithm. Four bits, values zero through fifteen, are sufficient for an array of thirty-two by thirty-two cells, which is a very large array for current technology.

4.1.1 MODIFIED S-VALUE

If we look at the present s-value algorithm, it produces a pattern of s- values

0 0 0 0 0

0 1 1 1 0

0 1 2 1 0

0 1 1 1 0

0 0 0 0 0.

Values of s continue to increase towards the center of the array. If we force any of the corner cells to become faulty, or if they all become faulty, the pattern will be

WITH RECONFIGURATION	WITHOUT RECONFIGURATION
X 0 0 0 X	X -1 0 -1 X
0 1 1 1 0	-1 0 1 0 -1
0 1 2 1 0	0 1 2 1 0
0 1 1 1 0	-1 0 1 0 -1
X 0 0 0 X	X -1 0 -1 X.

It is evident from this that the corner cells can never be used except in the case where the algorithm requires a single cell. The value cannot accurately predict the maximum square configuration, since some cells with a value of 1 will support a three by three square, while those at the corners will not. The failures of the corner cells do not change the values of s for internal cells.

A possibility to fix this problem is an algorithm that would produce s- value distributions such

as this.

1 1 1 1 1

1 3 3 3 1

1 3 5 3 1

1 3 3 3 1

1 1 1 1 1

The generation of these values is similar to the current algorithm, except a single bit is added to the s-value, and is called the augment bit. The algorithm would calculate s-values using these formulae-

1. All cells in the outside boundaries, and cells without a full complement of neighbors (4), such as cells bordering faulty cells where multiple faults exist (assuming the reconfiguration algorithm is available) would receive s-values of one(1). If the reconfiguration algorithm is not part of the logic, the cells isolating faulty cells would receive an s- value of -1. This value is fixed and will not be augmented.

2. Compute new s-values for internal cells using the minimum s-value of the cells neighbors plus 1.

1 1 1 1 1

1 2 2 2 1

1 2 3 2 1

1 2 2 2 1

1 1 1 1 1.

3. A cell would augment, or add 1 to its new s-value if any of the following conditions exist. Note that all s-values and augment bits refer to those computed in the previous steps 3 and 4 unless otherwise specified.

   a. Both the cells to the north and south have their augment bits set.

b. The augment bit to the north is set, and the s-value of the cell to the south is the same size or greater than the cells new s-value.

c. The augment bit to the south is set, and the s-value of the cell to the north is the same or larger than the cells new s-value.

d. The s-values of cells to the north and south are both at least as large as the new s-value calculated for the cell.

4. The "augment" bit is set in a cell if the new computed s-value for the cell is less than or equal to the old s-values of the eastern and western neighbors.

```
1   1+  1+  1+  1
1   2   2+  2   1
1   2   3   2   1
1   2   2+  2   1
1   1+  1+  1+  1
```

CYCLE	STEP 2	STEP 3	STEP 4
1	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1
	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1
	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1
	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1
	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1
2	1 1 1 1 1	1 1 1 1 1	1 1+1+1+1
	1 2 2 2 1	1 2 2 2 1	1 2 2 2 1
	1 2 2 2 1	1 2 2 2 1	1 2 2 2 1
	1 2 2 2 1	1 2 2 2 1	1 2 2 2 1
	1 1 1 1 1	1 1 1 1 1	1 1+1+1+1

3	1 1 1 1 1	1 1 1 1 1	1 1+1+1+1
	1 2 2 2 1	1 2 2 2 1	1 3 3 3 1
	1 2 3 2 1	1 2 3 2 1	1 3 3 3 1
	1 2 2 2 1	1 2 2 2 1	1 3 3 3 1
	1 1 1 1 1	1 1 1 1 1	1 1+1+1+1
4	1 1 1 1 1	1 1 1 1 1	1 1+1+1+1
	1 2 2 2 1	1 2 2 2 1	1 3 3+3 1
	1 2 3 2 1	1 2 3 2 1	1 3 3 3 1
	1 2 2 2 1	1 2 2 2 1	1 3 3+3 1
	1 1 1 1 1	1 1 1 1 1	1 1+1+1+1
5	1 1 1 1 1	1 1 1 1 1	1 1+1+1+1
	1 2 2 2 1	1 2 2 2 1	1 3 3+3 1
	1 2 4 2 1	1 2 4 2 1	1 3 5 3 1
	1 2 2 2 1	1 2 2 2 1	1 3 3+3 1
	1 1 1 1 1	1 1 1 1 1	1 1+1+1+1
6	1 1 1 1 1	1 1 1 1 1	1 1+1+1+1
	1 2 2 2 1	1 2 2 2 1	1 3 3+3 1
	1 2 4 2 1	1 2 4 2 1	1 3 5 3 1
	1 2 2 2 1	1 2 2 2 1	1 3 3+3 1
	1 1 1 1 1	1 1 1 1 1	1 1+1+1+1

This process continues each time step until equilibrium is reached. Equilibrium will be reached whenever s-values computed in step 2 using the output of the previously step 4 leads directly to a duplication of the previous step 2 values.

Using the above procedure, the s-value for any cell will never be larger than the size of the largest square of fault-free cells available centered on that cell. There may be times when the algorithm produces a value less than the maximum when faulty cells occur, but it will never be larger.

Proof. Assume a minimum sized array for which augmentation would be applicable.

1 1+ 1

1 2 1

1 1+ 1

This leads to an augmented distribution of

1 1+ 1

1 3 1

1 1+ 1

indicating cell(2,2) will support the growth of a three by three square.

Assume this is true for all values of s through (n-1). The minimum distribution of s_values for this configuration would be, showing only those values related to the augmentation of the cell of interest

(n-1) (n-1)+ (n-1)

(n-1) n (n-1)

(n-1) (n-1)+ (n-1)

Since cell with an s_value of (n-1) lie at the center of an (n-1) by (n-1) square of fault-free cells, then there are (n-2)/2 fault free columns to the left of column 1, and (n-2)/2 fault-free columns to the right of column 3. The total number of fault-free columns then is (n-2) + 3, to account for the rows and columns in the center of the pattern, which leads to a guaranteed neighborhood for the center cell(2,2) of (n+1) rows and columns. The s-value of cell(2,2) can be in-

cremented to a value of (n+1), leading to the pattern

(n-1) (n-1)+ (n-1)

(n-1) (n+1) (n-1)

(n-1) (n-1)+ (n-1) .

The remaining applications of the rules for augmentation of the s_value of cells to the north or south indicate an even larger fault-free area than the above example, so they will not be discussed here.

At first glance, the above algorithm appears to be asymmetrical, treating east/west neighbors differently from north/south neighbors. However, either pair of neighbors may be used to determine the augment possibilities for a cell. If we look at a fully augmented s-value array,

1 1+ 1+ 1+ 1

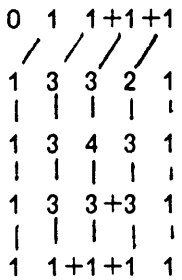
1 3 3+ 3 1

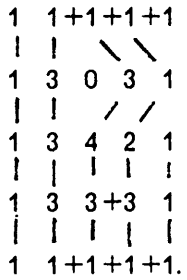
1 3 5 3 1

1 3 3+ 3 1

1 1+ 1+ 1+ 1

the s-value is symmetrical. If a cell becomes faulty, say cell(0,0) or cell(1,2), the s-value distributions become





Notice that even though we are using the north/south augment bits, if we were using the east/west neighbors, the results would be the same, since cell(2,1) would loose its augment indication, just as cell(1,2) did in our case.

4.2 PATTERN GROWTH

The current pattern growth algorithm [92] presents no apparent problems. However, a combination of two-dimensional and one-dimensional growth processes should be considered. A pattern could be grown from the seed until the pattern had reached its maximum number of rows. Growth could then proceed along each row using the one-dimensional algorithm. This allows large patterns to be grown closer to the edges of the array for more efficiency.

5.0 THE SIMULATION MODEL

This algorithm was simulated using Research Triangle Institute's ADAS (Architecture Design and Assessment System) with the CSIM software model simulator. This simulator combines graphics with a functional simulator. The use of graphics provides an excellent visual aid to debugging as well as a visual representation of what is happening with the functional simulator. The functional models for each of the different graph nodes is written in the C programming language.

5.1 THE GRAPHICS MODEL

5.1.1 CLOCK AND SPLIT NODES

The graphical model (Figure 21 on page 77) used in the simulation is a seven by seven matrix representing an execution array of forty-nine cells. This is the main body of the array. In addition, there is a clock node which provides timing tokens to initiate each of the operational

cycles of the array. These timing tokens are input to seven intermediate nodes , called split_, which provide tokens to each of the cell nodes in the execution array. The clock node produces one token each ten time increments, which is transmitted to the split_ nodes. The split_ nodes have firing delays set at zero, so they provide no time delay to the model. The firing delay of ten for the clock module has no physical meaning, and merely represents a convenient observation interval. Each of the cells in the execution array have firing delays of two, which is also without meaning.

The clock node has seven output arcs, one to each of the split_ nodes. Each of the split_ nodes has seven output arcs, each of which is connected to one node in the execution array in the same row as the split_ node.

5.1.2 NODES IN THE EXECUTABLE ARRAY

There are forty nine cell in the array of the simulation model. They are numbered from zero to forty eight, beginning with the upper left corner and increasing along the rows to cell number 48 in the lower left corner. Rows begin with cell numbers 0, 7, 14, 21, 28 and 35.

The nodes in the simulation array each have eleven input ports numbered zero through ten and ten output ports. The arcs delivering the clock tokens from the split_ nodes are connected to inport ten, with the other ten inports reserved for input from neighboring cells. All cell

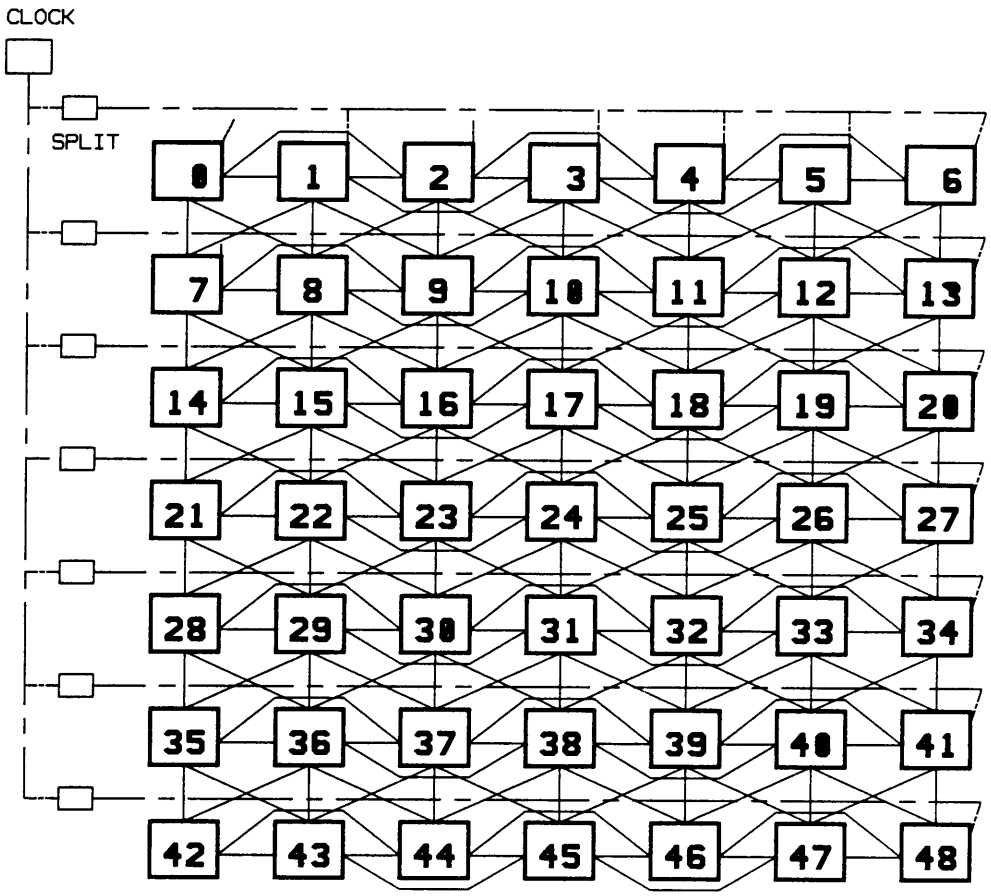


Figure 21. ADAS Simulation Model

outports are connected to neighboring cells. The port assignment for cell arcs is

0. Cell to the farwest (CELL 0)
1. Cell to the west (CELL 1)
2. Cell to the northwest (CELL 2)
3. Cell to the north (CELL 3)
4. Cell to the northeast (CELL 4)
5. Cell to the fareast (CELL 5)
6. Cell to the east (CELL 6)
7. Cell to the southeast (CELL 7)
8. Cell to the south (CELL 8)
9. Cell to the southwest (CELL 9)
10. Clock input

Since the port number three is assigned to be both the inport and the outport communicating with the cell to the north, the following pairing results between inports and outports. Inport 0 receives the output of outport 5 of the proper cell. Inport 1 is connected to an outport 6, inport 2 to outport 7, inport 3 to outport 8, inport 4 to outport 9, inport 5 to outport 0, inport 6 to outport 1, inport 7 to outport 2, inport 8 to outport 3, and inport 9 is connected to outport 4.

Arcs that make the above connections are routed to allow the graphical representation on the display resemble as nearly as possible a drawing of the active connections during simulation. Arcs have the default color of orange when the graph is initially drawn on the screen, but the color attribute is modified by the CSIM programs when simulation begins. During simulation, active arcs will have a color of red, and inactive arcs will be black. This gives the appearance almost identical to a hand-drawn representation of the array, and makes debugging of the model very easy.

Simulation is begun when a token from the split nodes reaches a cell inport. The node-user-text attribute of each cell is set to "any", which means any token received at any inport will start the execution of the cell. Inport nine is the only inport that receives a token by itself.

The token produce rates and consume rates on all other ports are set to one, which means that a cell will either be started by a single token from the clock, or by one token on each of the other nine inports. All tokens are consumed when received. Firing delays, the time allotted for cell execution is set to two time increments. Again, the time increments have no meaning, and were chosen only to make tracking the progress of the simulation easier.

Because of the setting of the node-user-text to allow each cell to treat each inport as a logical or gate, each cell will execute twice for each firing of the clock node. The cells will execute when the clock node produces a token at inport ten through the split nodes. This will force each cell to produce one token at each of the ten outports, which is then transmitted across the arcs to the inports of the neighboring cells. Each cell will now have tokens at all inports except ten. These tokens will be consumed and each cell will execute again. If the clock fires at Simulation Time 10, each cell will fire at Simulation Times 12 and 14. Simulation will run continuously until stopped by keyboard interrupt, or until a specified number of events has occurred.

At first glance, the graph may appear strange. To achieve the desired appearance, all execution cells are three by three grids in size. This enables routing arcs into a cell boundary without connecting to a port. Using this technique, all arcs either enter the center of one of the four edges of the node and go to a grid crossing before being connected to a port, or they enter the corner of a cell to a grid point and then to a port. This allows the graph to have a very symmetrical appearance when only the activated arcs are enabled.

There are no bi-directional arcs to simulate the method of communications in the array, so two arcs must be used. These two arcs are routed in the same space outside the cell boundary. This practice was found very useful for checking configurations and debugging the algorithm. When the simulator is renewing the screen graphics, if an arc from cell A to cell B is enabled, but not from B to A, the results will be obvious. The graphics is redrawn one cell and one arc at a time, by watching the graphics it is possible to tell when one cell has activated a port,

and the other has not. Either the arc from A to B will initially be colored red when cell A is drawn, and be changed to black when B is drawn, or the colors will change from black to red when the second cell is drawn.

The use of two arcs for each connection also allows the number of clock cycles used for the simulation to be reduced. Where the proposed cellular architecture uses six cycles of the clock to perform all operations, with this model we are able to reduce this to four cycles, transmitting from all cells during one cycle instead of two. Only two cycles of the clock node are used for each major cycle, because it is possible to allow the transmissions of data during a clock cycle to trigger the receipt of data and the computation cycles.

5.2 *THE SOFTWARE MODEL*

Each node in the hardware model above requires a software model be written. For this simulation, only three are required; clock, split, and cell. The models are contained in C files of those same names.

5.2.1 CLOCK AND SPLIT MODELS

The clock model does no functional operations. Therefore the model for this node contains no code.

The model for the split nodes also performs no functional operations, but this program is used to generate the necessary output to debug and verify operation of the model. The output generated by this program contains the current s-value of all cells, the current state, the status

of the multiple fault registers, and a combination of the connection-register (fault-register) and the enable-register. The enable register is a software register used only in the model. It contains the numbers of the ports in each cell that are enabled.

The output is activated by setting the node-user-integer of one of the split nodes to a non-zero value. Usually split0 is used for this purpose. The split_ nodes are used to generate the output because they execute only when all cells have completed execution. Having the cells produce the output would require changing the reporting cell each time the model size is changed.

The output is controlled by setting the user-integer in split0 to any number between one and seven. The output produced for each value of the integer is

1. Current s-value of all cells.

2. Current state and multiple-fault registers.

3. Both 1 and 2.

4. Connection-register and Enable-register

5. Both 1 and 4.

6. Both 2 and 4.

7. All three are reported.

5.2.2 THE CELL MODEL

The software mode for the cell contains the following registers.

- S-value
- Cell-state
- Global state
- Multiple-fault-register. This register contains individual software registers named double-fault-"x", where "x" is north, south, east, west, northeast, northwest, southeast or southwest. Each bit is programmed separately in the model.
- Fault-register. This is divided into north and south components for ease of programming, with each one being decoded to control the enables for that direction only.
- Current-State-Register This is the holding register that contains the state of all neighbors and the global state.
- Current-S-Register This register contains the current s-value of the neighboring cells.

5.2.2.1 INTERNAL TIMING

The timing of the cell operations is partitioned into four cycles, called 1A, 1B, 2A, and 2B. The cycles are controlled by setting the user integer on the outport 0 arc to either zero, one or two. Zero is used only during initialization to allow generation of normal connections between cells. A value of one or two signify cycles one or two. Cycles are further divided into A or B

sections by the status of inport 10. If this port is active, the model is in the A section, otherwise it is in the B section.

Coding for the four cycles is all contained within the same set of source code, without use of subroutines. However, the code for each cycle occupies contiguous lines within the source, and is clearly defined with comment statements.

CYCLE 1A: Cycle 1A in the model is a combination of cycles 1 and 2 in the cellular array. There are no multiplexed communications channels in the model, and therefore no cell priorities. All cells transmit their current s-value during this cycle, as well as part of the additional information required to implement the reconfiguration algorithm. This includes the contents of the multiple-fault-register. The definition of these bits will be given in the next section.

Cycle 1B: Execution of the cycle is begun when a cell receives input on ports other than ten after executing cycle 1A. All ports except ten will be active at this time since all outports on cells produce one token each time the cell fires. However, only that data that arrives on ports for which the enable term is set to a logical one will be read. All other input data will be ignored. The lower four bits of the input data is placed in a holding register. Input from one of the cells to the west will be placed in the lower four bits of the register, with those of the east, south and north occupying the next three bytes. The contents of this register is then examined to find the lowest value in any of the bytes. The new s-value for the cell is then calculated as one greater than this minimum value.

The remaining bits of the input are the contents of neighbors multiple fault registers. Each bit is examined on an individual basis, since the value of the bit, as well as the number of the inport determines the action to be taken.

If inport 1 is enabled, and the input contains bit 7 set, indicating the cell to the west has detected a multiple fault condition directly north of it, the double-fault-northwest bit is set in the cells multiple-fault- register. If bit 6 is set, the cell to the west has a multiple fault to the south, and the double-fault-southwest bit is set.

Inport three data is examined for bits 4 and 5, indicating the cell to the north has a multiple fault to the west or east. If either of these is set, double-fault northwest or northeast will be set. Inport 4 is also examined for bit 4 because the occurrence of side by side cell faults will create a multiple fault west in the cell to the east of the second fault, while the other multiple fault conditions will exist surrounding the western most failure. The cell logically south of the multiple fault west cell must recognize this to complete the isolation.

Inport six is examined for bits 6 and 7, and using the logic similar to that of inport one, may set double-fault southeast or southwest. Inport seven is examined for bit 1 to enable the setting of double-fault southwest.

Inport eight data is tested for bits 1 and 2, with the results being the setting of the double-fault southwest or southeast bits in the multiple- fault register.

CYCLE 2A: This cycle is used by all cells to transmit the value of their current state plus the contents of the fault-register and some additional control bits to implement reconfiguration. Only four bits were allotted for state representation in the model. These are transmitted in the lower four bits of the cycle 2A data. The next twelve bits contain the contents of the ten bits of the fault-register, a single bit to trigger the state reconfiguration, and one bit to force clear the array after a multiple fault.

The cellular array algorithm currently contains a clearing mechanism that uses the next-state logic to force an intermittent special state, and then a clearing of the state register after a fault. To minimize the coding in the next-state algorithm, a single bit is transmitted by a cell that

has either detected a multiple fault, or has received this bit from one of its neighbors. This has the effect of setting up sixteen special clearing states, and performs the same operation that the original algorithm performed, with much less code. This signal lasts only one clock cycle.

The logic for the fault-register data was explained in chapter 2. The remaining bit, called reconfigure, is generated as a one clock cycle signal whenever a fault is detected in the cell to the immediate west. This input is tested only on inport 1. If this bit is received, the logic of the next-state algorithm is disabled, the setting of the current-state register is disabled, and the next state for the cell is taken as the state of the cell to the west, the cell that transmitted the signal. This signal also is passed on to the eastern cells, and lasts only one clock cycle.

The next state logic is simulated using switch/case statements. Currently, only maintenance of the cleared state, and generation of a three by three active pattern from a seed state of 14 is programmed. Current state data is loaded into the current-state register when input from the neighboring cells. The byte locations for this register are; starting with byte zero; west, east, south, north and self. The contents of this register is used as the switch for calculating the next state.

5.2.2.2 *ADDITIONAL LOGIC*

The additional logic required to isolate multiple fault conditions, and force the array into pattern regeneration, is all executed during cycle 2B. This includes the double-fault north, south, east, and west bits in the multiple-fault register. These bits are set only by the contents of the fault-register.

The double-fault west bit is set if the contents of the fault-register indicate a faulty cell exists to the immediate west (port 1) and a faulty cell exists either to the east or the fareast.

Double-fault-north is set by faulty cells to the north and the northeast. Double-fault-south is set by faults to the south and southeast, while the double-fault-east bit is set by faulty conditions to the east and the fareast. The other four multiple-fault condition bits were set during cycle 1B.

5.2.2.3 FAULT INJECTION

Fault injection is accomplished by setting the node-user-integer attribute for a cell to zero. When a cell detects the integer equal to zero condition, it sets its s-value and its current state to zero. These values are transmitted to all neighboring cells. All enables for the outports are turn on and the faulty cell transmits continuous zero data to all cells to which it has communications paths. Only those cells testing the faulty cell will read the data however, and record the cell as faulty. Other cells must rely upon data from the original testing cells.

5.2.3 SIMULATION RESULTS

The model has been extensively tested. All single faulty cases have been tested, as well as a majority of the multiple fault cases. The remaining multiple fault cases not directly tested are variations on cases already tested. For example, The configuration of a cell with faults to the northwest and southwest is identical to that of a cell with faults to the northwest, southwest and fareast, since faults to the east in any of the three rows involving the cell do not affect its communications network. These cases were not fully tested.

6.0 CONCLUSIONS

We have analyzed systolic arrays, discussed their disadvantages and advantages, and the cellular array [92] which offers some solutions to some of these disadvantages. Problems exist with this algorithm however in the amount of time required for reconfiguration purposes. A possible solution to these problems has been presented which eliminates the lengthy reconfiguration process in the presence of only single faults in any row.

If we assume a seven by seven array, with a three by three active pattern imbedded, the clearing, neutralization and pattern growth process under the algorithm without reconfiguration typically would have required nearly thirty computation cycles. With a clock frequency of sixty MHz, this is approximately three thousand nano-seconds. With the reconfiguration algorithm logic in place, the reconfiguration after a single faulty cell is accomplished in four cycles. This converts to four hundred nano-seconds. The savings for single-fault reconfiguration is over eighty-five percent.

This savings is achieved through some added logic and additional transmission requirements to support the sharing of fault information. However, if the size of the cell under the original algorithm [92] is large, the added logic will be of little significance, and the additional yield and increased life of the chip may make the expenditures worthwhile.

The number of cells available for computation has been greatly increased. Under the original algorithm, faulty cells were isolated by assigning fault-free cells to that sole purpose. These isolation cells were given s- values of -1, which prohibited their use in pattern growth. Isolation is now done using switches, and the isolation cells are returned to the active part of the array. Previously, eight cells were used to isolate a faulty cell. None are used for that purpose with the new algorithm.

This research has yielded a reconfiguration algorithm for systolic arrays which achieves totally distributed control of the process without additional communications lines.

Also presented is an improved algorithm for determining the fault-free area surrounding a cell. This new algorithm defines the size of a square pattern that can be grown, rather than the size of a diamond. For most patterns, this should increase the efficiency of the pattern generation process, by providing shorter growth processes, and by allowing a higher survival possibility for arrays in which the patterns grown fit more nearly the square than the diamond.

6.1 ADDITIONAL WORK TO BE DONE

There is still much work to be done before attempting to fabricate an array using these algorithms. Methods must be found to isolate faulty cells in the execution plane. This may be accomplished by placing control cells with each of the switches in the execution plane. Because of the physical size expected in these cells, all possible ways to improve the efficiency of the pattern growth and reconfiguration processes must be investigated. The possibility of replacing the s-value [92] computation cycles with reconfiguration or array clearing and neutralization cycles when appropriate should be investigated. This may lead to still further improvements in efficiency by reducing the time required to reconfigure or regrow the pattern.

This alone could bring about a fifty percent time savings even without the reconfiguration algorithm, and with the reconfiguration algorithm, the savings may be as much as ninety percent. Additional reconfigurability may be available by slightly expanding the current communication network and seeking to reconfigure into spare rows and columns.

An ADAS simulation of the execution plane algorithms should be done, and if possible, the execution and control plane simulations combined into one simulation package should be done.

7.0 BIBLIOGRAPHY

1. H. T. Kung, "Why Systolic Architectures?",*COMPUTER*, Vol. 15, No. 1, pp. 37-46, January 1982.

2. L. Melkemi and M. Tchuente, "Complexity Of Matrix Product on a Class Of Otthoganally Connected Systolic Arrays," *IEEE Trans. on Computers*, Vol C-36, No. 5, pp. 615-619, May 1987.

3. J. A. B. Fortes and B. W. Wah, "Systolic Arrays- From Concept to Implementation," *Computer*, Vol. 20, No. 7, pp. 12-17, July 1987.

4. G. J. Li and B. W. Wah, "The Design Of Optimal Systolic Arrays," *IEEE Trans. on Computers*, Vol. C-34, No. 1, pp. 66-77, January 1985.

5. H. G. Yeh and H. Y. Yeh, "Implementation Of The Discrete Fourier Transform on 2-Dimensional Systolic Processors," *IEE Proc.-G Electronic Circuits And Systems*, Vol. 134, No. 4, pp. 181-185, August 1987.

6. B. W. LaMacchia and G. R. Redinbo, "RNS Digital Filtering Structures For Wafer-Scale Integration," *IEEE Journal on Selected Areas in Communications,* Vol. SAC-4, No. 1, pp. 67-86, Jan 1986.

7. L. Snyder. "Introduction to the Configurable Highly Parallel Computer," *Computer,* Vol. 15, No. 1, pp. 47-56, January 1982,

8. I. Koren and D. D. Pradhan, "Yield And Performance Enhancement Through Redundancy in VLSI And WSI Multiprocessor Systems," *Proc. Of The IEEE,* Vol. 74, No. 5, pp. 699-711, May 1986.

9. W. R. Moore and R. Mahat, "Fault-Tolerant Communications For Wafer- Scale Integration Of a Processor Array," *Microelectronics And Reliability,* Vol. 25, No. 2, pp. 292-294, 1985.

10. A. J. DeGroot, E. M. Johansson, J. P. Fitch, C. W. Grant, and R. S. Parker, "SPRINT- The Systolic Processor With A Reconfigurable Interconnection Network Of Transputers," *IEEE Trans. On Nuclear Science,* Vol. NS-34, No. 4,

11. F. T. Luk and S. Qiao, "Computing The CS-decomposition on Systolic Arrays," *SIAM Jour. Of Scientific And Statistical Computing,* Vol. 7, No. 4, pp. 1121-1125, Oct. 1986.

12. L. S. Haynes, R. L. Lau, D. P. Siewiorek, and D. W. Mizell, "A Survey Of Highly Parallel Computing," *Computer,* Vol. 15, No. 1, pp. 9-24, January 1982.

13. I. Koren and D. Pradhan, "Modeling The effect Of Redundancy on Yield And Performance Of VLSI Systems," *IEEE Trans. on Computers,* Vol. 36,No. 3, pp. 344-355, March 1986.

14. D. K. Pradhan, "Dynamically Restructurable Fault-Tolerant Processor Network Architectures," *IEEE Trans. on Computers,* Vol. C-34, No. 5, pp. 434-447, May 1985.

15. H. V. Jagadish, R. G. Mathews, T. Kailath, and J. A. Newkirk, "A Study Of Pipelining in Computing Arrays," *IEEE Trans. on Computers,* Vol. C-35, No. 5, pp. 431-439, May 1986.

16. J. L. Baer, "Computer Architecture" *Computer,* Vol. 17, No. 10. pp. 77-86 October 1984.

17. A. Y. Oruc, M. Y. Oruc and N. Balabaniar, "Reconfiguration Algorithms For Interconnection Networks," *IEEE Trans. on Computers,* Vol. C-34, No. 8, pp. 773-774, August 1985.

18. K. Hwang, "Multiprocessor Supercomputers For Scientific/Engineering Applications," *Computer,* Vol. 18, No. 6, pp. 57-73, June 1985.

19. W. T. Lin, C. Y Chin and C. Y. Ho, "Integrating Systolic Arrays into a Super-System," *Computer,* Vol. 20, No. 7, pp. 100-101, July 1987.

20. D. E. Fousler and R. Schreiber, "The SAXPY-1, A General-Purpose Systolic Computer," *Computer,* Vol. 20, No. 7, pp. 35-43, July 1987.

21. A. L. Rosenberg, "A Hypergraph Model For Fault-Tolerant VLSI Processor Arrays," *IEEE Trans. on Computers,* Vol. C-34, No. 6, pp. 578- 584, June 1985.

22. S. Yalamanchili and J. K Aggarbal, "Reconfiguration Stratagies For Parallel Architectures," *Computer,* Vol. 18, No. 12, pp 44-61, Dec. 1985.

23. A. Bojanczyk, R. P. Brent and H. T. Kung, "Numerically Stable Solution Of Dense Systems Of Linear Equations Using Mesh-Connected Processors," *Siam Jour. Of Science And Stitiscal Computing,* Vol. 5, No. 1, pp. 95-104, March 1984.

24. P. Comon and Yves Robert, "A Systolic Array For Computing BA^{-1}"
IEEE Trans. on Acoustics, Speach And Signal Processing, Vol. ASSP-35, No. 6, pp. 717-723, December 1987.

25. B. L. Drake, F. T. Luk, J. N. Speiser and J. L. Symanski, "SLAPP: A Systolic Linear Algebra Parallel Processor," *Computer*, Vol. 20, No. 7, pp 45-49, July 1987.

26. P. J. Eberlein, "On The Schur Decomposition Of a Matrix For Parallel Computation," *IEEE Trans. on Computers*, Vol. C-36, No. 2, pp. 167-174, February 1987.

27. L. Elden and R. Schreiber, "An Application Of Systolic Arrays to Linear Discrete Ill-Posed Problems," *Siam Jour. on Scientific And Statistical Computing*, Vol. 7, No. 3, pp. 892-903, Jul. 1986.

28. D. E. Heller and I. C. F. Ipsen, "Systolic Networks For Orthogonal Decomposition," *SIAM Jour. on Scientific And Statistical Computing*, Vol. 4, No. 2, pp. 261-269, June 1983.

29. J. Hoekstra, "Systolic Multiplier," *Electronics Letters*, Vol. 20, No. 24, pp. 995-996, Nov. 1984.

30. S. Y. Kung, "On Sypercomputing With Systolic/Wavefront Array Processors," *Proc. Of The IEEE*, Vol. 72, No. 7, pp. 867-884, Jul. 1985.

31. D. P. O'Leary, "Systolic Arrays For Matrix Transpose And Other Orderings," *IEEE Trans. on Computers*, Vol. C-36, No. 1, pp. 117-122, January 1987.

32. R. Schreiber, "Solving Eigenvalue And Singular Value Problems on an Undersized Systolic Array," *SIAM Journal of Scientific and Statistical Computing* , Vol. 7, No. 2, pp.441-451, April 1986.

33. U. Schweigelshohn and L. Thiele, "A Systolic Array For Cyclic-by-Rows Jacobi Algorithms," *Journal Of Parallel And Distributed Computing*, Vol. 4, No. 3. pp. 334-340, Jun. 1987.

34. D. S. Scott, M. T. Heath and R. C. Ward, "Parallel Block Jacobi Eigenvalue Algorithms Using Systolic Arrays," *Linear Algebra And Its Applications,* Vol. 77, pp. 345-355, May 1986.

35. J. A. E. Simons, "Systolic Convolution Array," *Electronics Letters,* Vol. 19, No. 21, p. 874, Oct. 1983.

36. M. K. Stojcev, I. Z. Milovanovic and Z. C. Radonjic, "Some Shifting Methods For Matrix Multiplication," *IEE Proc.-E Computers And Digital Techniques,* Vol. 132, No. 1, pp. 33-44, Jan. 1985.

37. P. J. Varman, I. V. Ramakrishnan and D. S. Fussell, "A Robust Matrix- Multiplication Array," *IEEE Trans. on Computers,* Vol. C-33, No. 10, pp. 919-922, October 1984.

38. P. J. Varman and J. V. Ramakrishnan, "Synthesis Of An Optimal Family Of Matrix Multiplication Algorithms on Linear Arrays," *IEEE Trans. on Computers,* Vol. C-35, No. 11, pp. 989-996, November 1985.

39. G. E. Bridges, W. Pries, R. D. McLeod, M. Ynnik, P. G. Gulak and H. C. Card, "Dual Systolic Architectures For VLSI Digital Signal Processing Systems," *IEEE Trans. on Computers,* Vol. C-35, No. 10, pp. 916- 923, October 1986.

40. J. A. B. Fortes and B. W. Wah, "Systolic Arrays - A Survey Of Seven Projects," *Computer,* Vol. 20, No. 7, pp. 91-103, July 1987.

41. J. M Jover and T. Kailath, "A Parallel Architecture Kalman Filter Measurement Oplate And Parameter Estimation,"" *Automatica,* Vol. 22, No. 1, pp. 43-57, Jan. 1986.

42. A. V. Kulkarni and D. W. L. Yen, "Systolic Processing And An Implementation For Signal And Image Processing," *IEEE Trans. on Computers,* vol C-31, No. 10, pp. 1000-1009, October 1982.

43. H. Lin, "New Systolic Array Design For Real-time Digital Signal Processing," *IEEE Trans. on Circuits And Systems,* Vol CAS-33, No. 7, pp. 673-676, Jul. 1986.

44. J. V. McCanny and J. G. McWhirter, "Some Systolic Array Developments in The United Kingdom," *Computer,* Vol. 20, No. 7, pp. 51-63, July 1987.

45. J. V. McCanny, R. A. Evans and J. G. McWhirter, "Use Of Unidirectional Data Flow in Bit-Level Systolic Array Chips," *Electronics Letters,* Vol 22, No. 10, pp. 540-541, May 1986.

46. J. G. McWhirter, D. Wood, K. Wood, R. A. Evans, J. V. McCanny and A. P. H. McCabe, "Multibit Convolution Using A Bit-Level Systolic Array," *IEEE Trans. on Circuits And Systems,* vol CAS-32, No. 1, pp. 95- 99, Jan. 1985.

47. J. P. Reilly, "Systolic Array Evaluation Of Polynomials With Application to Non-Linear Spectrum Estimation," *Electronics Letters,* vol 22, No. 24, pp. 1300-1302, November 1987.

48. C. L. Wang, C. H. Wei and S. H. Chen, "Improved Systolic Array For Linear Discrimanent Function Classifier," *Electronics Letters,* vol 22, No. 2, pp 85-86, Jan. 1986.

49. H. Derin and C. S. Won, "A Parallel Image Segmentation Algorithm Use Relaxation With Varying Neighborhoods And Its Mappint to Array Processors," *Computer Vision, Graphics And Image Processing,* Vol 40, No. 1, pp. 54-78, October 1987.

50. C. Guerra, "Systolic Algorithms For Local Operations on Images," *IEEE Trans. on Computers,* vol C-35, No. 1, pp. 73-77, January 1987.

51. S. Y. Lee and J. K Aggarwal, "Parallel 2-D Convolution on a Mush Connected Array Processor," *IEEE Trans. on Pattern Analysis And Machine Intellegence,* vol PAMI-9, No. 4, pp. 590-594, Jul. 1987.

52. L. M. Ni and A. K. Jain, "A VLSI Systolic Architecture For Pattern Clustering," *IEEE Trans on Pattern Analysis And Machine Intellegence*, Vol 7, No 1, pp. 80-89, Jan. 1985.

53. Z. C. Shih, G. H. Chen and R. C. T. Lee, "Systolic Algoritymns To Examine All Pairs Of Elements," *Communications Of The ACM*, vol 30, No. 2, pp. 161-167, Feb. 1987.

54. C. L. Wang, C. H. Wei and S. H. Chen, "Efficient Bit-Level Systolic Array For The Liniear Discriminant Function Classifier," *IEE Proc.-G Circuits And Systems*, vol 134, No. 5, pp 216-224, October 1987.

55. D. Gannon and J. VanPosendale, "On The Impact Of Communication Complexity On The Design Of Parallel Numerical Algorithms," *IEEE Trans. on Computers*, vol. C-33, No. 12, pp. 1180-1194, December 1984.

56. J. S. Ward, J. V. McCanny and J. G. McWhirter, "Bit Level Systolic Array Implementation Of The Winograd Fourier Transform Algorithm," *IEE Proc.-F Communication, Radar And Signal Processing*, vol 132, No. 6, pp 473-479; October 1985.

57. S. Ashtaputre and C. D. Savage, "Systolic Arrays With Embedded Tree Structures For Connectivity Problems," *IEEE Trans. on Computers*, vol C-34, No. 5, pp 483-484, May 1985.

58. S. Y. Kung, S. C. Lo and P. S. Lewis, "Optimal Systolic Design For The Transitive Closure And The Shortest Path Problems," *IEEE Trans. on Computers*, vol C-36, No. 5, pp 603-614, May 1987.

59. G. D. Lakhani, "An Improved Distribution Algorithm For Shortest Path Problem," *IEEE Trans. on Computers*, vol C-34, No. 9, pp 855- 856, September 1985.

60. B. P Sinha, B. B. Bhallacharya, S. Ghose and P. K. Srimani, "A Parallel Algorithm to Compute The Shortest Paths And Diameter Of A Graph And Its VLSI Implementation," *IEEE Trans. on Computers,* vol C- 35, No. 11, pp 1000-1004, November 1986.

61. A. W. Bojanczyk, "Systolic Implementation Of The Lattics Algorithm For Least Squares Linear Prediction Problems," *Linear Algebra And Its Applications* vol 77, pp 27-42, May 1986.

62. L. Elden and R. Schreiber, "A systolic Array For The Regularization Of Ill-Conditioned Least -Squares Problems with Triangular Toeplitz Matrix," *Linear Algebra And Its Applications* vol 77, pp 137-147, May 1986.

63. A. W. Bojanczyk and R. P. Brent, "A Systolic Algorithm For Extended GCD Computation," *Computers And Mathematics with Applications,* vol 14, No. 4, pp 233-238, Apr. 1987.

64. G. P. McKeown, "Iterated Interpolation Using A Systolic Array," *ACM Trans. on Mathematical Software,* vol 12, No. 2, pp 162-170, Jun. 1986.

65. C. W. Purdy and G. B. Purdy, "Integer Division In Linear Time With Bounded Fan-In," *IEEE Trans. on Computers,* Vol C-36, No. 5, pp 640-644, May 1987.

66. I. C. Wu, "A Fast 1-D Serial-Parallel Systolic Multiplier," *IEEE Trans. on Computers,* vol C-36, No. 10, pp 1243-1247, October 1987.

67. R. P. Brent and H. T. Kung, "Systolic VLSI Arrays For Polynomial GCD Computations," *IEEE Trans. on Computers,* vol C-33, No. 8, pp 731-736, August 1984.

68. S. H. Zak and K. Hwang, "Polynomial Division on Systolic Arrays," *IEEE Trans. on Computers,* vol C-34, No. 6, pp 577-578, June 1985.

69. W. Marwood and A. P. Clarke, "Therr Dimensional Systolic Architecture For Beamforming," *Eletronics Letters*, vol 21, No. 5, pp 212-213 Feb. 1985.

70. H. H. Chang, C. L. Nikias and A. N. Oenelsanopoulos, "Reconfigurable Systolic Array Implementation Of Quadratic Digital Filters," *IEEE Trans. on Circuits and Systems*, Vol CAS-33, No. 8, pp 845-848, Aug. 1986.

71. P. Cappello, "Gausian Elimitation On A Hyper-Cube Automaton," *Journal Of Parallel And Distributed Computing* Vol 4, No. 3, pp 288-308, Jun. 1987.

72. D. P. Lopresti, "P_NAC: A Systolic Array For Comparing Nuclei Acid Sequences," *Computer*, vol 20, No. 7, pp 98-99, July 1987.

73. A. K. Somani and V. K. Agarwal, "An Efficient Unsorted VLSI Dictionary Machine," *IEEE Trans. on Computers*, vol C-34, No. 9, pp 841- 852, September 1985.

74. M. K. Sridhar and R. Srinath, "On The Direct Parallel Solution Of Systems Of Linear Equations: New Algorithms And Systolic Structures," *Information Sciences*, vol 43, No. 1-2, pp 25-53, 1987.

75. C. S. Yeh, I. S. Reed and T. K. Truong, "Systolic Multipliers For Finite Fields GF.(2^m)," *IEEE Trans. on Computers*, vol C-33, No. 4, pp 357-360, April 1984.

76. J. A. Abraham, P. Banerjee, C.Y. Chen, W. K. Fuchs, S. Y. Kuo, A. L. Reddy, "Fault Tolerance Techniques for Systolic Arrays" *COMPUTER* Vol. 20, No. 7, pp. 65-75 July 1987

77. T. Leighton, C. Leiserson, "Wafer-Scale Integration Of Systolic Arrays," *IEEE Trans. On Computers* Vol. C-34, No. 5, pp. 448-461 May 1985.

78. P. Mazumder, "Evaluation Of On-Chip Static Interconnection Networks," *IEEE Trans. on Computers*, Vol. C-36, No. 3, pp. 365-369, March 1987.

79. R. Negrini, M. Sami, and R. Stephanelli, "Fault Tolerance Techniques For Array Structures Used in Super-Computing," *Computer*, Vol. 19, No. 2, pp. 78-86, Feb 1986.

80. R. A. Evans, J. V. McCanny, and K. W. Wood, "Wafer Scale Integration Based On Self-Organisation," in *WAFER SCALE INTEGRATION* eds. Jesshopeand Moore, Bristol: Adam-Hilger, 1986, pp. 101-112.

81. A. S. M Hassan and V. K. Agarwal, "A Fault-Tolerant Modular Architecture For Binary Trees," *IEEE Trans. On Computers*, Vol. C-35, No. 4, pp. 356-361, April 1986.

82. V. N Doniants, S. Lori, M. Pellagrino, E. L. Pi'il, and R. Stefanelli,"Fault-Tolerant Reconfigurable Processing Arrays Using Bi- Directional Switches," *Microprocessing And Microprogramming*, Vol. 14, No. 3-4, pp. 109-115, Oct-Nov, 1984.

83. R. Negrini, M. Sami, R. Stefanelli, "Fault-Tolerance Approaches For VLSI/WSI Arrays,"in *Proc IEEE Conference on Computers And Communications*, Phoenix, AZ. 1985.

84. M. Sami, "Reconfigurable Architectures For VLSI Arrays,"in Proc National Comp. Conf. AFIPS Los Angeles, CA. 1983 pp. 567-577

85. M. Sami, R. Stefanelli, "Reconfigurable Architectures For VLSI Processing Arrays," *Proc Of The IEEE*, Vol. 74, No. 5, pp. 712-722 May 1986.

86. R. M. Yanney and J. P. Hayes, "Distributed Recovery in Fault-Tolerant Multiprocessor Networks," *IEEE Trans. on Computers*, Vol C-35, No. 10, pp.871-879, Oct. 1986.

87. L. A. Shombert and D. P. Serwiorek, "Using Redundancy For Concurrent Testing And Repairing Of Systolic Arrays," *Proc. 17th. Int. Symp. on Fault Tolerant Computing,* pp. 244-249, July 1987.

88. W. Chen, P. B. Denyer, J. Manor, and D. Renshaw, "Fault Tolerant Wafer-Scale Architecture Using Large Crossbar Switch Arrays," in *Wafer Scale Integration,* eds. Jesshope and Moore, Bristol: Adam-Hilger, 1986, pp. 113-124.

89. T. Kondo, T. Nakashima, M. Aoki and T. Sudo, "An LSI Adaptive Array Processor", *IEEE Journal Of Solid-State Circuits,* Vol. SC-18, No. 2, pp. 147-156, April 1983.

90. C. L. Seitz, "Concurrent VLSI Architectures," *IEEE Trans. on Computers,* Vol. C-33, No. 12, pp. 1247-1265, December 1984.

91. F. Lombardi, R. Negrini, M. G. Sami and R. Stefanelli, "Reconfiguration Of VLSI Arrays: A Covering Approach," *Proc. 17th Int. Symp. on Fault Tolerant Computing,* pp. 251-256, July 1987.

92. R. Kumar,"A Fault-Tolerant Cellular Architecture," PH.D Dissertation, SU, Blacksburg, Virginia, 1984.

93. T. Y. Feng, "A Survey Of Interconnection Networks," *Computer,* Vol. 14, No. 12, pp. 12-27, December 1982.

94. H. L. Martin, "A Self-Reconfigurable Cellular Structure," PH.D Dissertation VPI&SU, Blacksburg, Virginia, 1980.

8.0 APPENDIX A CELL.C

```
/*************************************************************************
    FILE CELL.C
    THIS FILE CONTAINS THE EXECUTABLE CODE FOR CELLS IN THE EXECUTABLE
    ARRAY
*************************************************************************/

#include "cdefs.h"
int disable_0[144];              /*   DISABLES PORT 0 WHEN ADJACENT FAULTY
                                 CELLS OCCUR TO THE WEST          */

int double_fault_northeast[144];   /*   USED FOR CONNECTION CONTROL ONLY   */
int double_fault_southeast[144];   /*   NOW.  COULD ALSO BE USED FOR       */
int double_fault_northwest[144];   /*   PATTERN REGROWTH IF                */
int double_fault_southwest[144];   /*   NEEDED                             */
int enable_register[144];                        /* DEBUG USE ONLY */
int west_neighbor_state[144];      /*   PART OF THE HOLD_STATE REGISTER    */
int s_value[144], low_s[144];
int  local_state[144], global_state[144], outstate[144];
    /* OUTSTATE IS EITHER LOCAL STATE OR PRIORITY(DURING NEUTRALIZE CYCLE   */
int priority[144];               /* CELL PRIORITY IS THE CELL NUMBER      */
int clear_cell[144];
                  /* USED TO CLEAR AND NEUTRALIZE THE CELLS AFTER FAULTS,
                     BECAUSE OF THE LIMITED NUMBER OF COLORS IN ADAS */

/*ENABLE-X SIGNALS ARE USED TO CONTROL INPORTS AND OUTPORTS
  "0" INDICATES PORT IS INACTIVE!          */

int enable_0[144], enable_1[144], enable_2[144], enable_3[144], enable_4[144];
int enable_5[144], enable_6[144], enable_7[144], enable_8[144], enable_9[144];
int reconfigure[144];      /* CAUSES CELL TO TAKE ON WESTERN NEIGHBOR'S STATE*/

int double_fault_north[144];          /*      ISOLATE CELL DIRECTLY NORTH  */
int double_fault_south[144];
                          /* USED TO NEUTRALIZE AND REGROW PATTERN   */
int double_fault_east[144];
int double_fault_west[144];

int double_fault_north_delay[144];          /* USED FOR SPECIAL ENABLE CASES */
int double_fault_south_delay[144];
int fault_register[144];      /* TEN BITS DEFINING LOCATIONS OF FAULTY CELLS */
```

```
        /* FAULT_REGISTER_NORTH AND _SOUTH ARE MASKS ON FAULT_REGISTER   */
int fault_register_north[144];              /* CONTROLS ENABLES 2, 3 AND 4 */
int fault_register_south[144];              /* CONTROLS ENABLES 7, 8 AND 9 */

int neutralize[144];    /* CAUSES CELLS TO GATE IN PRIORITY INSTEAD OF STATE */

int hold_state[144];              /* CONTAINS STATES OF CELLS NEIGHBORS   */
int hold_svalue[144];             /* CONTAINS S_VALUES OF CELLS NEIGHBORS */

int double_fault[144]; /* COMMUNICATES TO NEIGHBORS MULTIPLE-FAULT CONDITIONS */
int double_fault1[144];                    /*  debug and reporting only */

int augment_north[144];       /*  USED TO INDICATE CELLS TO NORTH AND SOUTH */
int augment_south[144];       /*  HAVE AREAS TO EAST AND WEST AT LEAST AS LARGE
                          AS THEY ARE, I.E. S VALUE TO EAST AND WEST
                          G.T. OR EQUAL TO CELLS S-VALUE          */
int output_control[144];
int s_mark[144], s_mark_latch[144];/* USED TO MARK CELLS WHICH HAVE HOSTED THE
                            SEED STATE, AND TO HOLD NEIGHBORS S_MARK*/
int augment_s;

PROC cell(in0,in1,in2,in3,in4,in5,in6,in7,in8,in9,in10,
       out0,out1,out2,out3,out4,out5,out6,out7,out8,out9)

control in0[],in1[],in2[],in3[],in4[],in5[],in6[],in7[],in8[],
      in9[],out0[],out1[],out2[],out3[],out4[],out5[],out6[],
      out7[],out8[],out9[];
clok  in10[];
{
int   i, output, output_north, output_east, output_west, output_south;

/*  FOUR OUTPUT CONTROLS ARE NEEDED BECAUSE OF SEED MIGRATION          */

int cell_number;      /* CELLS ARE NUMBERED 0-48 IN SEQUENCE BY ROWS       */
/*************************************************************************
*
*        INITIALIZE CELL VALUES
*
*************************************************************************/
      cell_number = GetArcInt(IN10);
if (GetNTimes() == 1)   {
      SetUserInt(1);            /*CLEARS FAULTS FROM PREVIOUS SIMULATIONS */
      output_control[cell_number] = 0;
      s_mark[cell_number] = 0;
      s_mark_latch[cell_number] = 0;
      augment_north[cell_number] = 0;
      augment_south[cell_number] = 0;
      west_neighbor_state[cell_number] = 1;
      double_fault_north[cell_number] = 0;
      double_fault_south[cell_number] = 0;
      double_fault_east[cell_number] = 0;
      double_fault_west[cell_number] = 0;
      SetArcInt(OUT1,0);
      s_value[cell_number] = 1;  /*ESTABLISH INITIAL S_VALUES FOR ALL CELLS*/
      reconfigure[cell_number] = 0;
      global_state[cell_number] = 0;
```

```
      disable_0[cell_number] = 0;
      fault_register[cell_number] = 0;
      local_state[cell_number] = 1;
      priority[cell_number] = 0;
      neutralize[cell_number] = 0;
      clear_cell[cell_number] = 0;
      hold_state[cell_number] = 0x11111;
      double_fault[cell_number] = 0;
      double_fault_northwest[cell_number] = 0;
      double_fault_southwest[cell_number] = 0;
      double_fault_northeast[cell_number] = 0;
      double_fault_southeast[cell_number] = 0;     }
/*************************************************************************
     DEACTIVATE ALL PORTS AT BEGINNING OF EACH CYCLE
     PROGRAM ENABLES THOSE NEEDED FOR OPERATION
*************************************************************************/
      DeActivatePort(16); DeActivatePort(17);
      DeActivatePort(18); DeActivatePort(19);
      DeActivatePort(20); DeActivatePort(21);
      DeActivatePort(22); DeActivatePort(23);
      DeActivatePort(24); DeActivatePort(25);
if(cell_number == 0)
  if(GetUserFloat() == 0)
    augment_s = 0;
  else augment_s = 1;

/*  OUTPUT ARC_INTEGER "0"(16) IS USED TO KEEP TRACK OF "CLOCK PHASES"   */
/*  INPUT ARC_INTEGER "10" IS USED TO KEEP TRACK OF CELL "LOCATION"  */
/*  OUTPUT ARC_INT "1"(17) IS USED TO STORE THE CONNECTION FLAGS      */
/*  out5 ARC INTEGER USED TO DISABLE INCREMENTING S_VALUE FOR EDGE CELLS */

/*************************************************************************
*
*            CLOCK PHASE CONTROL PROCESS
*
*************************************************************************/

if(PortIsActive(10) == 1){
      if(GetNTimes() == 1)
        SetArcInt(OUT0,0);        /* INITIAL CLOCL CYCLE FOR SETUP ONLY */
      else {
        if (GetArcInt(OUT0) != 1)
          SetArcInt(OUT0,1);                    /* PHASES 1A AND 1B  */
        else {SetArcInt(OUT0,2);                  /* PHASES 2A AND 2B  */
  }
}
}
      /*   END OF CLOCK CONTROL PROCESS                  */

/*************************************************************************
FAULT INJECTION IS DONE BY SETTING THE CELL USER-INT ATTRIBUTE TO 0
*************************************************************************/

if((GetUserInt() == 0)&& (PortIsActive(IN10) == 1)){
      local_state[cell_number] = 0;
      s_value[cell_number] = 0;
```

```
        SetArcInt(OUT5,0);
}
/*****************************************************************************
 SEED INJECTION IS DONE BY SETTING THE NODE_USER_INTEGER TO 14 OR 15
*****************************************************************************/

if((PortIsActive(IN10) == 1) && (GetArcInt(OUT0) == 2)){
if((GetUserInt() >= 14)) {
   global_state[cell_number] = GetUserInt();
   SetUserInt(1);
 }
}
/*****************************************************************************
 EXTERNAL SEED INJECTION FOR TWO TEST CASES
   14 WILL PRODUCE A 3 BY 3 PATTERN OF
                                2 3 4
                                2 3 4
                                2 3 4,
   15   PRODUCES
                5 6 7 8
                9 A B 3
                D 3 B A
                9 8 7 6   PATTERN
*****************************************************************************/

/*****************************************************************
*
*                  OUTPUT  CONTROL
*
*****************************************************************/

/*****************************************************************
*
*                     CYCLE 1A
*
*****************************************************************/

/*****************************************************************************
 OUTPUT DURING CYCLE 1A IS: BITS  0-3 S_VALUE, 4-7= DOUBLE-FAULT CONDITIONS
                          8 = AUGMENT BIT, 9-10 = S_MARK
*****************************************************************************/

if(GetArcInt(OUT0) == 1){              /*  OUTPUT FOR CYCLE 1A = S VALUES  */
     output = s_value[cell_number]| (double_fault[cell_number] < < 4);

/***  OUTPUT THE AUGMENT BIT IS APPLICABLE          ************************/

if((((hold_svalue[cell_number]& 0xf)) > = s_value[cell_number])&&
   (((hold_svalue[cell_number] & 0xf0)> > 4) > = s_value[cell_number])){
if((double_fault_east[cell_number] == 0)&& (double_fault_west[cell_number]== 0))
  output = output | 0x100; }
  output = output | (s_mark[cell_number] < < 9);
  output_north = output;
  output_east  = output;
  output_west  = output;
  output_south = output; }
```

```
/*****************************************************************************
*
*                    CYCLE 2A
*
*****************************************************************************/
if(GetArcInt(OUT0) == 2){          /* OUTPUT FOR CYCLE 2A = CELL STATES  */
 if(neutralize[cell_number] == 0)
    outstate[cell_number] = local_state[cell_number];
 else outstate[cell_number] = priority[cell_number] | 0x80;
/* 0X80 USED BY RECEIVING CELL TO DEFINE NEUTRALIZE CYCLE INSTEAD OF STATE   */

if((neutralize[cell_number] == 0)&& (local_state[cell_number] != 0xc))

/* CELLS THAT INITIATE THE CLEARING AND NEUTRALIZATION PROCESSES HAVE
        LOCAL_STATE = 0XC*/

     output = outstate[cell_number] |
          ((fault_register[cell_number] ) < < 8)|
           (global_state[cell_number] < < 24);
else output = outstate[cell_number] | ((fault_register[cell_number]) < < 8);
     output_east = output;
     output_west = output;
     output_north = output;
     output_south = output;
if((global_state[cell_number] > = 14) && (local_state[cell_number] == 1)){
/*****************************************************************************
          CELL HAS THE SEED, BUT INSUFFICIENT S_VALUE
              ENTER SEED MIGRATION ROUTINE
*****************************************************************************/

  printf("CELL %d HAS THE SEED\n",cell_number);
 if(((global_state[cell_number] == 14) && (s_value[cell_number] > = 3))||
  (( global_state[cell_number] == 15) && (s_value[cell_number] > = 5))){
       local_state[cell_number] = global_state[cell_number];
       output = (output & 0xffffff00) | local_state[cell_number];
       output_east = output;
       output_west = output;
       output_north = output;
       output_south = output;}
/*****************************************************************
 IF S-VALUE LARGE ENOUGH, ENTER PATTERN GROWTH, ELSE TRANSMIT SEED
 SEED IS TRANSFERRED TO A CELL WITH HIGHER S-VALUE IN THE FOLLOWING
 PRIORITIES   SOUTH, EAST, WEST, AND NORTH.  IF NO HIGHER S-VALUE IS FOUND
 SEED IS PASSED TO THE SOUTH.
 ( BOTTOM TWO ROWS , DEFAULT IS TO THE NORTH IF S_VALUE TEST FAILS )
 NOTE:  LATER ADDITIONS SHOULD INCLUDE TEST TO INSURE THE CELL TO WHICH
 THE SEED IS TO BE PASSED EXISTS.....
*****************************************************************/
 else { global_state[cell_number] = 0;

/*****************************************************************
PRIORITY AS PROGRAMMED  S MARK = 0 AND S IS GREATER
                        S MARK = 0 AND S IS EQUAL
                        S MARK = 0 AND S IS LESS
                        S MARK = 1 AND S IS GREATER
```

```
                S MARK = 1 AND S IS EQUAL
                S MARK = 1 AND S IS LESS
*************************************************************************/

if(((s_mark_latch[cell_number] & 0x30) == 0) &&      /*S-MARK OF SOUTH IS 0  */
  (((hold_svalue[cell_number]& 0x0f00) > > 8) > s_value[cell_number]))
{   output_control[cell_number] = 1;
      s_mark[cell_number] = 1; }

else if(((s_mark_latch[cell_number] & 0x0c) == 0) &&
  (((hold_svalue[cell_number]& 0x0f0) > > 4) > s_value[cell_number]))
{   output_control[cell_number] = 2;
      s_mark[cell_number] = 1; }

else if(((s_mark_latch[cell_number] & 0xc0) == 0) &&
  (((hold_svalue[cell_number]& 0x0f000) > > 12) > s_value[cell_number]))
{   output_control[cell_number] = 3;
      s_mark[cell_number] = 1; }

else if(((s_mark_latch[cell_number] & 0x03) == 0) &&
  (((hold_svalue[cell_number]& 0x0f)) > s_value[cell_number]))
{   output_control[cell_number] = 4;
      s_mark[cell_number] = 1; }

else if(((s_mark_latch[cell_number] & 0x30) == 0) &&
  (((hold_svalue[cell_number]& 0x0f00) > > 8) == s_value[cell_number]))
{   output_control[cell_number] = 1;
      s_mark[cell_number] = 1; }

else if(((s_mark_latch[cell_number] & 0x0c) == 0) &&
  (((hold_svalue[cell_number]& 0x0f0) > >4) == s_value[cell_number]))
{   output_control[cell_number] = 2;
      s_mark[cell_number] = 1; }

else if(((s_mark_latch[cell_number] & 0xc0) == 0) &&
  (((hold_svalue[cell_number]& 0x0f000) > > 12) == s_value[cell_number]))
{   output_control[cell_number] = 3;
      s_mark[cell_number] = 1; }

else if(((s_mark_latch[cell_number] & 0x03) == 0) &&
  (((hold_svalue[cell_number]& 0x0f)) == s_value[cell_number]))
{   output_control[cell_number] = 4;
      s_mark[cell_number] = 1; }

else if(((s_mark_latch[cell_number] & 0x30) == 0) &&
  ((((hold_svalue[cell_number]& 0x0f00) > > 8) + 1) == s_value[cell_number]))
{   output_control[cell_number] = 1;
      s_mark[cell_number] = 1; }

else if(((s_mark_latch[cell_number] & 0x0c) == 0) &&
  ((((hold_svalue[cell_number]& 0x0f0) > >4) + 1) == s_value[cell_number]))
{   output_control[cell_number] = 2;
      s_mark[cell_number] = 1; }

else if(((s_mark_latch[cell_number] & 0xc0) == 0) &&
```

```
  ((((hold_svalue[cell_number]& 0x0f000) > > 12) + 1) = = s_value[cell_number]))
{   output_control[cell_number] = 3;
      s_mark[cell_number] = 1; }

else if(((s_mark_latch[cell_number] & 0x03) = = 0) &&
  (((hold_svalue[cell_number]& 0x0f) + 1) = = s_value[cell_number]))
{   output_control[cell_number] = 4;
      s_mark[cell_number] = 1; }
else if(((s_mark_latch[cell_number] & 0x30) = = 0) &&
  ((((hold_svalue[cell_number]& 0x0f00) > > 8) + 2) = = s_value[cell_number]))
{   output_control[cell_number] = 1;
      s_mark[cell_number] = 1; }

else if(((s_mark_latch[cell_number] & 0x0c) = = 0) &&
  ((((hold_svalue[cell_number]& 0x0f0) > > 4) + 2) = = s_value[cell_number]))
{   output_control[cell_number] = 2;
      s_mark[cell_number] = 1; }

else if(((s_mark_latch[cell_number] & 0xc0) = = 0) &&
  ((((hold_svalue[cell_number]& 0x0f000) > > 12) + 2) = = s_value[cell_number]))
{   output_control[cell_number] = 3;
      s_mark[cell_number] = 1; }

else if(((s_mark_latch[cell_number] & 0x03) = = 0) &&
  (((hold_svalue[cell_number]& 0x0f) + 2) = = s_value[cell_number]))
{   output_control[cell_number] = 4;
      s_mark[cell_number] = 1; }

else if(((s_mark_latch[cell_number] & 0x30) = = 0x10) &&
  (((hold_svalue[cell_number]& 0x0f00) > > 8) > s_value[cell_number]))
{   output_control[cell_number] = 1;
      s_mark[cell_number] = 2; }

else if(((s_mark_latch[cell_number] & 0x0c) = = 4) &&
  (((hold_svalue[cell_number]& 0x0f0) > > 4) > s_value[cell_number]))
{   output_control[cell_number] = 2;
      s_mark[cell_number] = 2; }

else if(((s_mark_latch[cell_number] & 0xc0) = = 0x40) &&
  (((hold_svalue[cell_number]& 0x0f000) > > 12) > s_value[cell_number]))
{   output_control[cell_number] = 3;
      s_mark[cell_number] = 2; }

else if(((s_mark_latch[cell_number] & 0x03) = = 1) &&
  (((hold_svalue[cell_number]& 0x0f)) > s_value[cell_number]))
{   output_control[cell_number] = 4;
      s_mark[cell_number] = 2; }

else if(((s_mark_latch[cell_number] & 0x30) = = 0x10) &&
  (((hold_svalue[cell_number]& 0x0f00) > > 8) = = s_value[cell_number]))
{   output_control[cell_number] = 1;
      s_mark[cell_number] = 2; }

else if(((s_mark_latch[cell_number] & 0x0c) = = 4) &&
  (((hold_svalue[cell_number]& 0x0f0) > > 4) = = s_value[cell_number]))
{   output_control[cell_number] = 2;
```

```
      s_mark[cell_number] = 2; }

else if(((s_mark_latch[cell_number] & 0xc0) == 0x40) &&
  (((hold_svalue[cell_number]& 0x0f000) >> 12) == s_value[cell_number]))
{   output_control[cell_number] = 3;
      s_mark[cell_number] = 2; }

else if(((s_mark_latch[cell_number] & 0x03) == 1) &&
  (((hold_svalue[cell_number]& 0x0f)) == s_value[cell_number]))
{   output_control[cell_number] = 4;
      s_mark[cell_number] = 2; }
else if(((s_mark_latch[cell_number] & 0x30) == 0x10) &&
  ((((hold_svalue[cell_number]& 0x0f00) >> 8)+1) == s_value[cell_number]))
{   output_control[cell_number] = 1;
      s_mark[cell_number] = 2; }

else if(((s_mark_latch[cell_number] & 0x0c) == 4) &&
  ((((hold_svalue[cell_number]& 0x0f0) >>4)+1) == s_value[cell_number]))
{   output_control[cell_number] = 2;
      s_mark[cell_number] = 2; }

else if(((s_mark_latch[cell_number] & 0xc0) == 0x40) &&
  ((((hold_svalue[cell_number]& 0x0f000) >> 12)+1) == s_value[cell_number]))
{   output_control[cell_number] = 3;
      s_mark[cell_number] = 2; }

else if(((s_mark_latch[cell_number] & 0x03) == 1) &&
  (((hold_svalue[cell_number]& 0x0f)+1) == s_value[cell_number]))
{   output_control[cell_number] = 4;
      s_mark[cell_number] = 2; }
else if(((s_mark_latch[cell_number] & 0x30) == 0x10) &&
  ((((hold_svalue[cell_number]& 0x0f00) >> 8)+2) == s_value[cell_number]))
{   output_control[cell_number] = 1;
      s_mark[cell_number] = 2; }

else if(((s_mark_latch[cell_number] & 0x0c) == 4) &&
  ((((hold_svalue[cell_number]& 0x0f0) >>4)+2) == s_value[cell_number]))
{   output_control[cell_number] = 2;
      s_mark[cell_number] = 2; }

else if(((s_mark_latch[cell_number] & 0xc0) == 0x40) &&
  ((((hold_svalue[cell_number]& 0x0f000) >> 12)+2) == s_value[cell_number]))
{   output_control[cell_number] = 3;
      s_mark[cell_number] = 2; }

else if(((s_mark_latch[cell_number] & 0x03) == 1) &&
  (((hold_svalue[cell_number]& 0x0f)+ 2) == s_value[cell_number]))
{   output_control[cell_number] = 4;
      s_mark[cell_number] = 2; }


/*************************************************************************
              ALL S-MARKS ARE 2,   NO WAY OUT
************************************************************************* */

else output_control[cell_number] = 0;
```

```
/*****************************************************************************
                        SEED OUTPUT CONTROL
***************************************************************************** */

switch (output_control[cell_number]){
/* OUTPUT CONTROL IS USED TO DETERMINE WHICH OF THE NIEGHBORING
   CELLS WILL RECEIVE THE SEED                                          */
case 0: {printf("seed is in an irrecoverable state system halt\n");}

case 1: {
    output_south = output ;                    /* SOUTH GETS GLOBAL STATE*/
    output_east  = output & 0x0ffffff;      /* OTHERS DO NOT GET GLOBAL ST. */
    output_north = output_east;
    output_west  = output_east;       break;}
case 2: {
    output_east = output;
    output_south  = output & 0x0ffffff;
    output_north = output_south;
    output_west  = output_south;      break;}
case 3: {
    output_north = output ;
    output_east  = output & 0x0ffffff;
    output_south = output_east;
    output_west  = output_east;      break;}
case 4: {
    output_west = output ;
    output_east  = output & 0x0ffffff;
    output_north = output_east;
    output_south  = output_east;      break;}
} } } }

/*****************************************************************************
*
*                    INPUT CONTROL
*
*****************************************************************************/

if(GetNTimes() != 1) {            /*   INPUT DISABLED ON INITIALIZATION   */
if((GetArcInt(OUT0) == 1)&&(PortIsActive(IN10) == 0)&&
     (local_state[cell_number] != 0)){

/*****************************************************************
*               CYCLE 1 B
*          INPUT NEIGHBORS S_VALUES
*
*      ALL TOKENS WILL BE CONSUMED, BUT DATA WILL BE ACCEPTED ONLY
*      FROM THOSE PORTS THAT ARE ACTIVATED WITH ENABLE_
*
*****************************************************************/
augment_north[cell_number] = 0;
augment_south[cell_number] = 0;
hold_svalue[cell_number] = 0;
/*****************************************************************************
```

```
      S_VALUE AND AUGMENTS WILL BE RECALCULATED EACH CYCLE
*************************************************************************/
/*************************************************************************
  PLEASE NOTE THAT IN ALL CASES FOR INPUT CYCLES, THE CODE IS WRITTEN
  TO INPUT EITHER PORT 0 OR PORT 1,  ONE OF PORTS 2, 3, AND 4, EITHER
  PORT 5 OR PORT 6, AND ONE OF PORTS 7, 8, OR 9
*************************************************************************/

if ((PortIsActive(IN0) = =1)&&(enable_0[cell_number] = =1) )
{  hold_svalue[cell_number] = in0[0] & 0xf ;
   s_mark_latch[cell_number] = (in0[0] & 0x600) > > 9; }

/*************************************************************************
  FOR AN INPUT TO BE UTILIZED, THE PORT MUST BE ACTIVE,
  AND THE PORT MUST BE SOFTWARE "ENABLED"
*************************************************************************/

else if ((PortIsActive(IN1) = =1)&&(enable_1[cell_number] = =1) )
{ if(((in1[0] & 0x80) = = 0x80)&&(double_fault_northwest[cell_number] = = 0))
       + +double_fault_northwest[cell_number] ;
  if(((in1[0] & 0x40) = = 0x40)&&(double_fault_southwest[cell_number] = = 0))
       + +double_fault_southwest[cell_number] ;
  hold_svalue[cell_number] = in1[0] & 0xf;
   s_mark_latch[cell_number] = (in1[0] & 0x600) > > 9; }

 if ((PortIsActive(IN2) = =1)&&(enable_2[cell_number] = =1) )
{ if((in2[0] & 0x100) = = 0x100) + +augment_north[cell_number];
  hold_svalue[cell_number] = hold_svalue[cell_number] | ((in2[0] & 0xf) < < 12);
   s_mark_latch[cell_number] =
         s_mark_latch[cell_number] | ((in2[0] & 0x600) > > 3); }

else if ((PortIsActive(IN3) = =1)&&(enable_3[cell_number] = =1))
{  if((in3[0] & 0x100) = = 0x100) + +augment_north[cell_number];
if(((in3[0] & 0x10) = = 0x10)&&(double_fault_northwest[cell_number] = = 0))
              + +double_fault_northwest[cell_number] ;
  if(((in3[0] & 0x20) = = 0x20)&&(double_fault_northeast[cell_number] = = 0))
             + +double_fault_northeast[cell_number] ;
  hold_svalue[cell_number] = hold_svalue[cell_number] | ((in3[0] & 0xf) < < 12);
   s_mark_latch[cell_number] = s_mark_latch[cell_number] |
                          ((in3[0] & 0x600) > > 3); }

else if ((PortIsActive(IN4) = =1)&&(enable_4[cell_number] = =1) )
{  if((in4[0] & 0x100) = = 0x100) + +augment_north[cell_number];
if((in4[0] & 0x10) = = 0x10)          double_fault_northwest[cell_number] =1;

  hold_svalue[cell_number] = hold_svalue[cell_number] | ((in4[0] & 0xf) < < 12);
   s_mark_latch[cell_number] =  s_mark_latch[cell_number] |
                          ((in4[0] & 0x600) > > 3); }

if ((PortIsActive(IN5) = =1 )&&(enable_5[cell_number] = =1) )
{

  hold_svalue[cell_number] = hold_svalue[cell_number] | ((in5[0] & 0xf) < < 4);
   s_mark_latch[cell_number] =  s_mark_latch[cell_number] |
                          ((in5[0] & 0x600) > > 7); }
```

```
else if ((PortIsActive(IN6) = =1 )&&(enable_6[cell_number] = =1) )
{ if(((in6[0] & 0x80) = = 0x80)&&(double_fault_northeast[cell_number] = = 0))
        + +double_fault_northeast[cell_number] ;
  if(((in6[0] & 0x40) = = 0x40)&&(double_fault_southeast[cell_number] = = 0))
           + +double_fault_southeast[cell_number] ;

  hold_svalue[cell_number] = hold_svalue[cell_number] | ((in6[0] & 0xf) < < 4);
   s_mark_latch[cell_number] =  s_mark_latch[cell_number] |
                          ((in6[0] & 0x600) > > 7); }

if ((PortIsActive(IN7) = =1 )&&(enable_7[cell_number] = =1) )
{ if((in7[0] & 0x100) = = 0x100) + +augment_south[cell_number];
if(((in7[0] & 0x10) = = 0x10)&&(double_fault_southwest[cell_number] = = 0))
        + +double_fault_southwest[cell_number] ;

  hold_svalue[cell_number] = hold_svalue[cell_number] | ((in7[0] & 0xf) < < 8);
   s_mark_latch[cell_number] =  s_mark_latch[cell_number] |
                          ((in7[0] & 0x600) > > 5); }

else if ((PortIsActive(IN8) = =1)&&(enable_8[cell_number] = =1) )
{ if((in8[0] & 0x100) = = 0x100) + +augment_south[cell_number];
if(((in8[0] & 0x10) = = 0x10)&&(double_fault_southwest[cell_number] = = 0))
          + +double_fault_southwest[cell_number] ;
  if(((in8[0] & 0x20) = = 0x20)&&(double_fault_southeast[cell_number] = = 0))
             + +double_fault_southeast[cell_number] ;

  hold_svalue[cell_number] = hold_svalue[cell_number]|((in8[0] & 0xf) < < 8);
   s_mark_latch[cell_number] =  s_mark_latch[cell_number] |
                          ((in8[0] & 0x600) > > 5); }

else if ((PortIsActive(IN9) = =1)&&(enable_9[cell_number] = =1) )
{ if((in9[0] & 0x100) = = 0x100) + +augment_south[cell_number];
  hold_svalue[cell_number] = hold_svalue[cell_number]|((in9[0] & 0xf) < < 8);
   s_mark_latch[cell_number] =  s_mark_latch[cell_number] |
                          ((in9[0] & 0x600) > > 5); }
if(priority[cell_number] != 0)
{ s_mark_latch[cell_number] = 0;
  s_mark[cell_number] = 0;  }
/*************************************************************************
      SET UP THE S_MARK_LATCH TO PREVENT TRANSPORTING SEED
      TO A NON-EXISTANT CELL  FROM A BOUNDARY CELL
*************************************************************************/
if((cell_number < 7)||(double_fault_north[cell_number] != 0))
  s_mark_latch[cell_number] = s_mark_latch[cell_number] | 0x80;
if(((cell_number % 7) = = 0)||(double_fault_west[cell_number] != 0))
  s_mark_latch[cell_number] = s_mark_latch[cell_number] | 0x2;
if((cell_number > 41)||(double_fault_south[cell_number] != 0))
  s_mark_latch[cell_number] = s_mark_latch[cell_number] | 0x20;
if((((cell_number + 1) % 7) = = 0)||(double_fault_east[cell_number] != 0))
  s_mark_latch[cell_number] = s_mark_latch[cell_number] | 0x8;
if(((cell_number + 1) % 7) = = 0){
  if(enable_7[cell_number] != 0)
       s_mark_latch[cell_number] = s_mark_latch[cell_number] | 0x20;
  if(enable_4[cell_number] != 0)
       s_mark_latch[cell_number] = s_mark_latch[cell_number] | 0x80;}
if(((cell_number + 6) % 7) = = 0)
```

```
  if(enable_0[cell_number] != 0)
        s_mark_latch[cell_number] = s_mark_latch[cell_number] | 0x2;
if(((cell_number + 2) % 7) == 0)
  if(enable_5[cell_number] != 0)
        s_mark_latch[cell_number] = s_mark_latch[cell_number] | 0x8;


/*************************************************************************
*
*            CALCULATE NEW S_VALUE   CYCLE 1B
*
*************************************************************************/
if((GetArcInt(OUT5) ==4)){
        /* DISABLE INCREMENTING S_VALUE ON BOUNDARY  CELLS */

low_s[cell_number] = hold_svalue[cell_number] & 0xf;
if(((hold_svalue[cell_number] & 0x00f0) > > 4) < low_s[cell_number])
    low_s[cell_number] =(hold_svalue[cell_number] & 0x00f0) > > 4;

if(((hold_svalue[cell_number] & 0x0f00) > > 8) < low_s[cell_number])
    low_s[cell_number] =(hold_svalue[cell_number] & 0x0f00) > > 8;

if(((hold_svalue[cell_number] & 0xf000) > > 12) < low_s[cell_number])
    low_s[cell_number] =(hold_svalue[cell_number] & 0xf000) > > 12;

  s_value[cell_number] = low_s[cell_number] + 1;
/*DETERMINE THE LOWEST INPUT VALUE OF S, AND INCREMENT.  THIS IS NEW
  UN_AUGMENTED S-VALUE FOR THIS CELL  */

if(augment_s != 0){
/* S-VALUE AUGMENTATION IS ALLOWED IF THE INFORMATION FROM THE CELLS
   TO THE NORTH AND SOUTH INDICATE THE PRESENCE OF SUFFICIENT FAULT-FREE
   SPACE TO SUPPORT A SQUARE PATTERN OF A SIZE LARGER THAN THAT INDICATED
   BY THE NORMAL "DIAMOND" S_VALUE                                    */
if( (s_value[cell_number] > = 2)&&

  ((augment_north[cell_number]= = 1)&& (augment_south[cell_number]= =1))||

  ((augment_north[cell_number]= =1) &&
        (((hold_svalue[cell_number]& 0xf00) > >8) > = s_value[cell_number]))||

  ((augment_south[cell_number]= =1) &&
        (((hold_svalue[cell_number]& 0xf000) > >12) > = s_value[cell_number]))||

      ((((hold_svalue[cell_number]& 0xf000) > >12) > = s_value[cell_number])&&
        (((hold_svalue[cell_number]& 0xf00) > >8) > = s_value[cell_number]))
 ){
if((s_value[cell_number] % 2)= = 0)
    + +s_value[cell_number];
}

if(double_fault[cell_number] != 0)
  s_value[cell_number] = 1;                        /*DEBUG ONLY */
} } }
/*************************************************************************
*                CYCLE 2B
```

```
            INPUT OF NEIGHBORS CURRENT STATES OR PRIORITIES
                    AND FAULT CONDITIONS
*             LOCAL_STATES ARE LIMITED TO 4 BITS CURRENTLY
*******************************************************************************/

if ((GetArcInt(OUT0) == 2)&&(PortIsActive(IN10) == 0 )){/* CYCLE = 2B      */
  if(global_state[cell_number] == 0) neutralize[cell_number] = 0;
  reconfigure[cell_number] = 0;
  hold_state[cell_number] = (hold_state[cell_number] & 0x01111)|
                        local_state[cell_number] < < 16;
/* TRANSFER LOCAL STATE TO HOLDING REGISTER, AND CLEAR NEIGHBORS STATES  */
  if(enable_0[cell_number] == 1){/* READ AND PROCESS INPUT FROM PORT 0 */
     if ((PortIsActive(IN0) == 1)&& (in0[0] != 0xd)){
   if((in0[0] & 0xf000000) != 0)
      global_state[cell_number] = (in0[0] & 0xf000000) > > 24;
    if((in0[0] & 0x80) == 0)            /* LOAD STATE REGISTERS  */
       hold_state[cell_number] = (hold_state[cell_number]& 0xffff0) |
                            (in0[0] & 0xf);

    else {                              /* LOAD PRIORITY REGISTERS */
         neutralize[cell_number] = 1;
      if((in0[0] & 0x7f) > priority[cell_number]){
        priority[cell_number] = in0[0] & 0x7f;
        local_state[cell_number] = 1;
        global_state[cell_number] = 0; }}

/*  READ FAULTY CELL LOCATION IN NEXT INSTRUCTION SEQUENCE      */

     if (((in0[0] & 0x8f) == 0)||((in0[0] & 0x900) > 0))
       fault_register[cell_number] =
           fault_register[cell_number] | 0x8;
     if (((in0[0] & 0x8f) == 0)&& (double_fault_west[cell_number] == 0))
       { disable_0[cell_number] = 1;
      + +double_fault_west[cell_number] ;}
   /*  TWO FAULTY CELLS TO THE  LEFT --- PATTERN MUST BE REGROWN   */

/*******************************************************************************
   REMAINING IN0 CODE IS INPUT OF THE FAULT REGISTER INFORMATION
    ALL PORT INPUT SEQUENCES ARE SIMILAR
*******************************************************************************/
      if(((in0[0] & 0x2400) > = 1))
       fault_register[cell_number] =
           fault_register[cell_number] | 0x20;
     if(((in0[0] & 0x1200) > = 1))
       fault_register[cell_number] =
            fault_register[cell_number] | 0x10;
   if((in0[0] & 0x100000) == 0x100000)
        clear_cell[cell_number] = 1;
}}

else if(enable_1[cell_number] == 1){/* READ AND PROCESS INPUT FROM PORT 1 */
    if((PortIsActive(IN1) == 1)&& (in1[0] != 0xd)){
  if((in1[0] & 0xf000000) != 0)
     global_state[cell_number] = (in1[0] & 0xf000000) > > 24;

   if((in1[0] & 0x80) == 0)             /* LOAD STATE REGISTERS  */
```

```
      hold_state[cell_number] = (hold_state[cell_number]& 0xfff0) |
                                (in1[0] & 0xf);
   else { /*LOAD NEUTRALIZATION INFORMATION       */
        neutralize[cell_number] = 1;
    if((in1[0] & 0x7f) > priority[cell_number]){
      priority[cell_number] = in1[0] & 0x7f;
      local_state[cell_number] = 1;
      global_state[cell_number] = 0; }}
    if ((in1[0] & 0x8f) == = 0 ){
      reconfigure[cell_number] = 1;
     fault_register[cell_number] =
         fault_register[cell_number] | 0x1; }

   if((in1[0] & 0x200000) > =1) reconfigure[cell_number] = 1;

     if((in1[0] & 0x900) > = 1)
      fault_register[cell_number] =
         fault_register[cell_number] | 0x8;

     if((in1[0] & 0x1200) > = 1)
      fault_register[cell_number] =
         fault_register[cell_number] | 0x10;

     if((in1[0] & 0x2400) > = 1)
      fault_register[cell_number] =
         fault_register[cell_number] | 0x20;

     if (reconfigure[cell_number]= = 0)
      west_neighbor_state[cell_number] = (in1[0] & 0x0f);
     if((in1[0] & 0x100000) = = 0x100000)
        clear_cell[cell_number] = 1;

}}

if(enable_2[cell_number] = = 1){

   if ((PortIsActive(IN2) = =1)&&(in2[0] != 0xd)){
  if((in2[0] & 0xf000000) != 0)
     global_state[cell_number] = (in2[0] & 0xf000000)> > 24;

   if((in2[0] & 0x80) = = 0)              /* LOAD STATE REGISTERS  */
      hold_state[cell_number] = (hold_state[cell_number]& 0xf0fff) |
                                ((in2[0] & 0xf)< < 12);
   else {
        neutralize[cell_number] = 1;
    if((in2[0] & 0x7f) > priority[cell_number]){
      priority[cell_number] = in2[0] & 0x7f;
      local_state[cell_number] = 1;
      global_state[cell_number] = 0; }}
    if ((in2[0] & 0x8f) = = 0){
     fault_register[cell_number] =
         fault_register[cell_number] | 0x10;
     if((fault_register[cell_number] & 0x102) != 0)
       double_fault_northwest[cell_number] = 1;
 }
     if((in2[0] & 0x100000) = = 0x100000)
```

```
        clear_cell[cell_number] = 1;

      if ((in2[0] & 0x4000) == 0x4000)
         fault_register[cell_number] =
           fault_register[cell_number] | 0x2;
      if ((in2[0] & 0x8000) == 0x8000)
         fault_register[cell_number] =
           fault_register[cell_number] | 0x100;
      if ((in2[0] & 0x900) != 0)
         fault_register[cell_number] =
           fault_register[cell_number] | 0x10;
} }

else  if(enable_3[cell_number] == 1) {
  if ((PortIsActive(IN3) == 1)&& (in3[0] != 0xd)){
  if((in3[0] & 0xf000000) != 0)
     global_state[cell_number] = (in3[0] & 0xf000000) > > 24;
   if((in3[0] & 0x80) ==0)                    /* LOAD STATE REGISTERS  */
      hold_state[cell_number] = (hold_state[cell_number]& 0xf0fff) |
                          ((in3[0] & 0xf) < < 12);
   else {
        neutralize[cell_number] = 1;
    if((in3[0] & 0x7f) > priority[cell_number]){
      priority[cell_number] = in3[0] & 0x7f;
      local_state[cell_number] = 1;
      global_state[cell_number] = 0; }}

   if ((in3[0] & 0x8f) == 0 )
{      fault_register[cell_number] =
          fault_register[cell_number] | 0x2;
}
     if((in3[0] & 0x100000) == 0x100000)
        clear_cell[cell_number] = 1;

   if((in3[0] & 0xc000) != 0)
      fault_register[cell_number] =
        fault_register[cell_number] | 0x100;
   if((in3[0] & 0x900) != 0)
      fault_register[cell_number] =
        fault_register[cell_number] | 0x10;
} }

else if(enable_4[cell_number] == 1){
   if ((PortIsActive(IN4) == 1)&&(in4[0] != 0xd)){
  if((in4[0] & 0xf000000) != 0)
     global_state[cell_number] = (in4[0] & 0xf000000) > > 24;

   if((in4[0] & 0x80) ==0)                    /* LOAD STATE REGISTERS  */
     hold_state[cell_number] =
          (hold_state[cell_number]& 0xf0fff) | ((in4[0] & 0xf) < < 12);
   else {
        neutralize[cell_number] = 1;
    if((in4[0] & 0x7f) > priority[cell_number]){
      priority[cell_number] = in4[0] & 0x7f;
      local_state[cell_number] = 1;
      global_state[cell_number] = 0; }}
```

```
      if((in4[0] & 0x8f) == 0){
      fault_register[cell_number] =
          fault_register[cell_number] | 0x100;
}
      if((in4[0] & 0x100000) == 0x100000)
         clear_cell[cell_number] = 1;

      if ((in4[0] & 0x100) == 0x100)
         fault_register[cell_number] =
           fault_register[cell_number] | 0x2;
      if ((in4[0] & 0x800) != 0)
         fault_register[cell_number] =
           fault_register[cell_number] | 0x10;   /* 18-dec-87*/
      if ((in4[0] & 0xc000) != 0)
         fault_register[cell_number] =
           fault_register[cell_number] | 0x100;
    }}

if(enable_5[cell_number] == 1){
  if ((PortIsActive(IN5) == 1 )&&(in5[0] != 0xd)){
  if((in5[0] & 0xf000000) != 0)
     global_state[cell_number] = (in5[0] & 0xf000000) >> 24;

   if((in5[0] & 0x80) ==0)                    /* LOAD STATE REGISTERS  */
      hold_state[cell_number] =
       (hold_state[cell_number]& 0xfff0f) | ((in5[0] & 0xf) << 4);
   else {
       neutralize[cell_number] = 1;
     if((in5[0] & 0x7f) > priority[cell_number]){
       priority[cell_number] = in5[0] & 0x7f;
       local_state[cell_number] = 1;
       global_state[cell_number] = 0; }}

    if((in5[0] & 0x8f) == 0){
      fault_register[cell_number] =
          fault_register[cell_number] | 0x80;
 }
     if ((in5[0] & 0xc000) > 0)
      fault_register[cell_number] =
          fault_register[cell_number] | 0x80;

     if ((in5[0] & 0x20400) > 0)
      fault_register[cell_number] =
          fault_register[cell_number] | 0x200;

     if ((in5[0] & 0x10200) > 0)
      fault_register[cell_number] =
          fault_register[cell_number] | 0x100;

     if ((in5[0] & 0x100000) == 0x100000)
        clear_cell[cell_number] = 1;
} }

 else if(enable_6[cell_number] == 1){
    if ((PortIsActive(IN6) == 1 )&&(in6[0] != 0xd)){
```

```
  if((in6[0] & 0xf000000) != 0)
     global_state[cell_number] = (in6[0] & 0xf000000) > > 24;
   if((in6[0] & 0x80) = =0)                    /* LOAD STATE REGISTERS  */
     hold_state[cell_number] =
          (hold_state[cell_number]& 0xfff0f) | ((in6[0] & 0xf) < < 4);
  else {
       neutralize[cell_number] = 1;
    if((in6[0] & 0x7f) > priority[cell_number]){
      priority[cell_number] = in6[0] & 0x7f;
      local_state[cell_number] = 1;
      global_state[cell_number] = 0; }}

   if((in6[0] & 0x8f) = = 0)
{     fault_register[cell_number] =
         fault_register[cell_number] | 0x40; }

   if((in6[0] & 0xc000) > = 1)
     fault_register[cell_number] =
         fault_register[cell_number] | 0x80;

    if ((in6[0] & 0x20400) > 0)
     fault_register[cell_number] =
         fault_register[cell_number] | 0x200;

    if ((in6[0] & 0x10200) > 0)
     fault_register[cell_number] =
         fault_register[cell_number] | 0x100;

    if ((in6[0] & 0x100000) = = 0x100000)
       clear_cell[cell_number] = 1;
}   }

if(enable_7[cell_number] = = 1){
   if((PortIsActive(IN7) = = 1)&& (in7[0] != 0xd )){
  if((in7[0] & 0xf000000) != 0)
     global_state[cell_number] = (in7[0] & 0xf000000) > > 24;
 if((in7[0] & 0x80) = =0)                    /* LOAD STATE REGISTERS  */
      hold_state[cell_number] =
           (hold_state[cell_number]& 0xff0ff) | ((in7[0] & 0xf) < < 8);
   else {
        neutralize[cell_number] = 1;
     if((in7[0] & 0x7f) > priority[cell_number]){
       priority[cell_number] = in7[0] & 0x7f;
       local_state[cell_number] = 1;
       global_state[cell_number] = 0; }}

   if((in7[0] & 0x8f) = = 0){
      fault_register[cell_number] =
          fault_register[cell_number] | 0x200;}
     if((in7[0] & 0x100000) = = 0x100000)
        clear_cell[cell_number] = 1;
     if ((in7[0] & 0x100) = = 0x100)
        fault_register[cell_number] =
          fault_register[cell_number] | 0x4;
     if ((in7[0] & 0xc000) != 0)
        fault_register[cell_number] =
```

```
            fault_register[cell_number] | 0x200;
        if ((in7[0] & 0x800) != 0)
            fault_register[cell_number] =
              fault_register[cell_number] | 0x20;
} }

else if(enable_8[cell_number] == 1){
  if((PortIsActive(IN8) == 1)&& (in8[0] != 0xd)){
  if((in8[0] & 0xf000000) != 0)
     global_state[cell_number] = (in8[0] & 0xf000000) > > 24;
 if((in8[0] & 0x80) = =0)                    /* LOAD STATE REGISTERS  */
      hold_state[cell_number] =
          (hold_state[cell_number]& 0xff0ff) | ((in8[0] & 0xf) < < 8);
   else {
        neutralize[cell_number] = 1;
     if((in8[0] & 0x7f) > priority[cell_number]){
       priority[cell_number] = in8[0] & 0x7f;
       local_state[cell_number] = 1;
       global_state[cell_number] = 0; }}

 if((in8[0] &0x8f) == 0)
{     fault_register[cell_number] =
          fault_register[cell_number] | 0x4;      }
     if((in8[0] & 0x100000) == 0x100000)
        clear_cell[cell_number] = 1;
 if((in8[0] & 0x900) != 0)
      fault_register[cell_number] =
          fault_register[cell_number] | 0x20;
 if((in8[0] & 0xc000) != 0)
      fault_register[cell_number] =
          fault_register[cell_number] | 0x200;
} }

else if(enable_9[cell_number] == 1){
  if((PortIsActive(IN9) == 1)&&(in9[0] != 0xd)){
  if((in9[0] & 0xf000000) != 0)
     global_state[cell_number] = (in9[0] & 0xf000000) > > 24;
   if((in9[0] & 0x80) = =0)              /* LOAD STATE REGISTERS  */
      hold_state[cell_number] =
              (hold_state[cell_number]& 0xff0ff) | ((in9[0] & 0xf) < < 8);
   else {
        neutralize[cell_number] = 1;
     if((in9[0] & 0x7f) > priority[cell_number]){
       priority[cell_number] = in9[0] & 0x7f;
       local_state[cell_number] = 1;
       global_state[cell_number] = 0; }}

      if ((in9[0] & 0x8f) == 0){
       fault_register[cell_number] =
          fault_register[cell_number] | 0x20;
     if (((fault_register[cell_number] & 0x204) != 0)&&
          ( double_fault_southwest[cell_number] == 0))
          + +double_fault_southwest[cell_number];    }

      if((in9[0] & 0x100000) == 0x100000)
         clear_cell[cell_number] = 1;
```

```
      if ((in9[0] & 0x4000) == 0x4000)
         fault_register[cell_number] =
           fault_register[cell_number] | 0x4;
      if ((in9[0] & 0x900) != 0)
         fault_register[cell_number] =
           fault_register[cell_number] | 0x20;
      if ((in9[0] & 0x8000) != 0)
         fault_register[cell_number] =
           fault_register[cell_number] | 0x200;
}}

 if ((local_state[cell_number] == 0xc)||((local_state[cell_number] == 1)&&
            (priority[cell_number] == 0)))
  { clear_cell[cell_number] = 0;
    neutralize[cell_number] = 0; }
if(local_state[cell_number] == 0xc)
  hold_state[cell_number] = 0xc1111;
if(double_fault1[cell_number] != 0){
  if((hold_state[cell_number]& 0xf) == 0)
    hold_state[cell_number] = hold_state[cell_number] | 1;
  if((hold_state[cell_number]& 0xf0) == 0)
    hold_state[cell_number] = hold_state[cell_number] | 0x10;
  if((hold_state[cell_number]& 0xf00) == 0)
    hold_state[cell_number] = hold_state[cell_number] | 0x100;
  if((hold_state[cell_number]& 0xf000) == 0)
    hold_state[cell_number] = hold_state[cell_number] | 0x1000;
}
/*****************************************************************************
*
*                CALCULATE NEW STATE
*
*****************************************************************************/
/*****************************************************************************
*
*      DATA READ DURING CYCLE 2B OF THE CLOCK IS PLACED IN A HOLDING
*      REGISTER CALLED "hold_state" IN THE FOLLOWING MANNER
*      FROM LEFT TO RIGHT
*
*      POSITION 1 IS THE CELLS OWN STATE
*      POSITION 2 IS THE NORTH NEIGHBORS STATE
*      POSITION 3 IS THE SOUTH NEIGHBORS STATE
*      POSITION 4 IS THE EAST NEIGHBORS STATE
*      POSITION 5 IS THE WEST NEIGHBORS STATE
*
*****************************************************************************/
if((GetUserInt() != 0)&& (neutralize[cell_number] == 0)){
   switch ( hold_state[cell_number])  {

 case 0x11111: { if(global_state[cell_number] == 0)
              local_state[cell_number] = 1; break;}
 case 0xc1111: { local_state[cell_number] = 0xc; break;}

/*****************************************************************************
  THE FOLLOWING CODE REPRESENTS THE NEXT-STATE LOOKUP TABLE FOR
  STATE TRANSISTIONS IN PATTERN GROWTH.  TO ADD ADDITIONAL PATTERNS
```

```
 THE TABLE MUST BE GENERATED FROM  THE INITIAL SEED ( IT MAY BE NECESSARY TO
 USE ADDITIONAL STATES WITH VALUES HIGHER THAN 16, SO THE STATE TABLE FOR ALL
 PATTERNS MUST BE RE-CODED  AND COLORS ASSIGNED FOR OTHER THAN STATES 0-16

 A PATTERN IS GROWN BY SETTING THE LOCAL-STATE TO A UNIQUE VALUE WHEN THE
 PROPER S-VALUE IS DETECTED, AND GENERATING THE CASE: STATEMENTS FOR THE
 LOOK-UP TABLE.  ADDITIONS SHOULD BE ADDED AFTER THE 4 BY 4 CODE TO PREVENT
 CONFUSION OVER DUPLICATE PATTERN CASES.  THE COMPILER WILL ANOUNCE DUPLICATE
 CASE PATTERNS
*****************************************************************************/

/* THE FOLLOWING STATEMENTS GROW A
                2 3 4
                2 3 4
                2 3 4 PATTERN FROM A SEED OF "14"   */

 case 0x11114: { ;}
 case 0x11121: { ;}
 case 0x11211: { ;}
 case 0x11311: { ;}
 case 0x11411: { ;}
 case 0x12111: { ;}
 case 0x13111: { ;}
 case 0x14111: { local_state[cell_number] = 1;
            global_state[cell_number] = 0; break;}
 case 0x12131: { local_state[cell_number] = 2; break;}
 case 0x111e1: { local_state[cell_number] = 2; break;}
 case 0x11231: { local_state[cell_number] = 2; break;}
 case 0x21131: { local_state[cell_number] = 2; break;}
 case 0x21231: { local_state[cell_number] = 2; break;}
 case 0x22231: { local_state[cell_number] = 2; break;}
 case 0x22131: { local_state[cell_number] = 2; break;}
 case 0x31311: { local_state[cell_number] = 3; break;}
 case 0x31342: { local_state[cell_number] = 3; break;}
 case 0x33142: { local_state[cell_number] = 3; break;}
 case 0x33111: { local_state[cell_number] = 3; break;}
 case 0x33342: { local_state[cell_number] = 3; break;}
 case 0xe1111: { local_state[cell_number] = 3; break;}
 case 0x11e11: { local_state[cell_number] = 3; break;}
 case 0x1e111: { local_state[cell_number] = 3; break;}
 case 0x11413: { local_state[cell_number] = 4; break;}
 case 0x44413: { local_state[cell_number] = 4; break;}
 case 0x14113: { local_state[cell_number] = 4; break;}
 case 0x41113: { local_state[cell_number] = 4; break;}
 case 0x41413: { local_state[cell_number] = 4; break;}
 case 0x44113: { local_state[cell_number] = 4; break;}
 case 0x1111e: { local_state[cell_number] = 4; break;}
/*****************************************************************

  produce
        5678
        9AB3
        43BA
        9876 pattern  FROM A "SEED " OF 15
*****************************************************************/
 case 0xf1111: {local_state[cell_number] = 0xb; break; }
```

```
case 0x11f11: {local_state[cell_number] = 0xb; break; }
case 0x1f111: {local_state[cell_number] = 7; break; }
case 0x1111f: {local_state[cell_number] = 0xa; break; }
case 0x111f1: {local_state[cell_number] = 0x3; break; }
case 0xbb7a3: {local_state[cell_number] = 0xb; break; }
case 0xb1b11: {local_state[cell_number] = 0xb; break; }
case 0xa111b: {local_state[cell_number] = 0xa; break; }
case 0x7b111: {local_state[cell_number] = 7; break; }
case 0x71b11: {local_state[cell_number] = 7; break; }
case 0x311b1: {local_state[cell_number] = 0x3; break; }
case 0x11a1b: {local_state[cell_number] = 0x3; break; }
case 0x11b11: {local_state[cell_number] = 7; break; }
case 0x113b1: {local_state[cell_number] = 0xa; break; }
case 0x11131: {local_state[cell_number] = 4; break; }
case 0x13171: {local_state[cell_number] = 8; break; }
case 0x1a117: {local_state[cell_number] = 6; break; }
case 0x11317: {local_state[cell_number] = 8; break; }
case 0x11961: {local_state[cell_number] = 5; break; }
case 0x51961: {local_state[cell_number] = 5; break; }
case 0x61a75: {local_state[cell_number] = 6; break; }
case 0x6a117: {local_state[cell_number] = 6; break; }
case 0x11a71: {local_state[cell_number] = 6; break; }
case 0x61a71: {local_state[cell_number] = 6; break; }
case 0x7b168: {local_state[cell_number] = 7; break; }
case 0x71b86: {local_state[cell_number] = 7; break; }
case 0x81317: {local_state[cell_number] = 8; break; }
case 0x83171: {local_state[cell_number] = 8; break; }
case 0x83179: {local_state[cell_number] = 8; break; }
case 0x114a1: {local_state[cell_number] = 9; break; }
case 0x914a1: {local_state[cell_number] = 9; break; }
case 0x14181: {local_state[cell_number] = 9; break; }
case 0x94181: {local_state[cell_number] = 9; break; }
case 0x954a1: {local_state[cell_number] = 9; break; }
case 0xa361b: {local_state[cell_number] = 0xa; break; }
case 0xa63b9: {local_state[cell_number] = 0xa; break; }
case 0xa13b1: {local_state[cell_number] = 0xa; break; }
case 0xb7b3a: {local_state[cell_number] = 0xb; break; }
case 0x3a8b4: {local_state[cell_number] = 0x3; break; }
case 0x31a1b: {local_state[cell_number] = 0x3; break; }
case 0x38a1b: {local_state[cell_number] = 0x3; break; }
case 0x41131: {local_state[cell_number] = 4; break; }
case 0x49931: {local_state[cell_number] = 4; break; }
case 0x11113: {; }    /* BOUNDARY CELLS NOT IN THE ACTIVE PATTERN    */
case 0x1111a: {; }
case 0x11116: {; }
case 0x16111: {; }
case 0x17111: {; }
case 0x18111: {; }
case 0x19111: {; }
case 0x11191: {; }
case 0x11141: {; }
case 0x11151: {; }
case 0x11511: {; }
case 0x11611: {; }
case 0x11711: {; }
case 0x11811: {; }
```

```
 case 0x11118: { local_state[cell_number] = 1;
/*    printf("state of %d = 1 from case statement\n",cell_number);*/
          global_state[cell_number] = 0; break; }
/* DEBUGGING TOOL- USEFUL WHEN CODING STATE TRANSISTION TABLES, BUT CREATES
   OUTPUT CLUTTER WHEN IN NORMAL USE  PRINTS OUTPUT FOR RECONFIGURING CELLS*/

default: { if(local_state[cell_number] == 1)
         global_state[cell_number] = 0;
/*************************************************************************
 IF CODE DOES NOT EXIST FOR A PATTERN, THE CELL IS DEFAULTED TO QUIESCENT
*************************************************************************/
} } }

/*************************************************************************
   IF RECONFIGURE = 1 AND CLEAR CELL = 0, RECONFIGURE
*************************************************************************/
  if ((reconfigure[cell_number] == 1)&&
            ((fault_register[cell_number] & 0x1000) != 0x1000))
   if((GetUserInt() != 0)){
        local_state[cell_number] = west_neighbor_state[cell_number];}

/*********************************************************************
IF CLEAR = 1, CLEAR THE CELL TO QUIESCENT STATE OF "1"
*********************************************************************/
if((clear_cell[cell_number] == 1)&&(local_state[cell_number] != 0))
  {
          local_state[cell_number] = 1;
          priority[cell_number] = 0;
          neutralize[cell_number] = 0;
          global_state[cell_number] = 0;
  fault_register[cell_number] =
       fault_register[cell_number] | 0x1000;  /*TRANSMIT CLEAR CELL SIGNAL*/
    }}}
         /*      END NEW STATE ALGORITHM        */

/*************************************************************************
*         ARRAY CLEARING AND NEUTRALIZATION CODE
*                  TIMING IS 2B
*
*************************************************************************/
/* THIS CODE IS EXECUTED BY CELLS USED TO ISOLATE THE WESTERN FAULTY CELLS   */
if((GetArcInt(OUT0) == 2) && (PortIsActive(IN10) != 1)){

if(((fault_register[cell_number] & 0x204) == 0x204)){

 if(double_fault_south[cell_number] < 40)
     ++double_fault_south[cell_number];
       switch (double_fault_south[cell_number]){
   case  1: {;}
   case  2: {if(global_state[cell_number] != 0){  /* CELL WITH GLOBAL STATE
                                        INITIALIZES CLEAR AND
                                        NEUTRALIZE          */
           local_state[cell_number] = 0xc;
           clear_cell[cell_number] = 1;}  break;}
   case  3: {clear_cell[cell_number]  = 0;  break;}
   case 10: {if(global_state[cell_number] != 0){/* TRANSMIT PRIORITY    */
```

```
            priority[cell_number] = cell_number;
            neutralize[cell_number] = 1;} break;}
  case 25: {neutralize[cell_number] = 0;    break;}
  case 26: {if(local_state[cell_number] == 0xc)
          clear_cell[cell_number] = 1;    break;}
  case 27: {clear_cell[cell_number] = 0;      break;}
  case 28: {priority[cell_number] = 0;        break;}
  case 39: {if(local_state[cell_number] == 0xc)/* BECOME SEED CELL IF
                                    STILL HOLDING GLOBAL STATE*/
         local_state[cell_number] = 1;    break;}
}}

if(((fault_register[cell_number] & 0xc0) == 0xc0)||
     (double_fault_east[cell_number] != 0 )){/* DETECT DOUBLE FAULTY TO EAST*
/

 if(double_fault_east[cell_number] < 40)
    ++double_fault_east[cell_number];
      switch (double_fault_east[cell_number]){
  case  1: {;}
  case  2: {if(global_state[cell_number] != 0){
          local_state[cell_number] = 0xc;
          clear_cell[cell_number] = 1;}  break;}
  case  3: {clear_cell[cell_number]  = 0;  break;}
  case 10: {if(global_state[cell_number] != 0){
            priority[cell_number] = cell_number;
            neutralize[cell_number] = 1;} break;}
  case 25: {neutralize[cell_number] = 0;    break;}
  case 26: {if(local_state[cell_number] == 0xc)
          clear_cell[cell_number] = 1;    break;}
  case 27: {clear_cell[cell_number] = 0;      break;}
  case 28: {priority[cell_number] = 0;        break;}
  case 39: {if(local_state[cell_number] == 0xc)
         local_state[cell_number] = 1;    break;}
}}

if(((fault_register[cell_number] & 0x102) == 0x102)){
 if(double_fault_north[cell_number] < 40)
    ++double_fault_north[cell_number];

/* INITIATE CLEARING AND NEUTRALIZATION PROCESS COUNTER    */

      switch (double_fault_north[cell_number]){
  case  1: {;}
  case  2: {if(global_state[cell_number] != 0){
          local_state[cell_number] = 0xc;
          clear_cell[cell_number] = 1;}     break;}
  case  3: {clear_cell[cell_number]  = 0;     break;}
  case 10: {if(global_state[cell_number] != 0){
            priority[cell_number] = cell_number;
            neutralize[cell_number] = 1;}  break;}
  case 25: {neutralize[cell_number] = 0;      break;}
  case 26: {if(local_state[cell_number] == 0xc)
          clear_cell[cell_number] = 1;    break;}
  case 27: {clear_cell[cell_number] = 0;      break;}
  case 28: {priority[cell_number] = 0;        break;}
```

```
  case 39: {if(local_state[cell_number] == 0xc)
           local_state[cell_number] = 1;    break;}
}}
if(((fault_register[cell_number] & 0x41) == 0x41)||
              ((fault_register[cell_number] & 0x81) == 0x81)||
                ( double_fault_west[cell_number] != 0)){
 if(double_fault_west[cell_number] < 40)
    ++double_fault_west[cell_number];
      switch (double_fault_west[cell_number]){
  case  1: {;}
  case  2: {if(global_state[cell_number] != 0){
           local_state[cell_number] = 0xc;
           clear_cell[cell_number] = 1;}  break;}
  case  3: {clear_cell[cell_number]  = 0;  break;}
  case 10: {if(global_state[cell_number] != 0){
             priority[cell_number] = cell_number;
             neutralize[cell_number] = 1;} break;}
  case 25: {neutralize[cell_number] = 0;    break;}
  case 26: {if(local_state[cell_number] == 0xc)
            clear_cell[cell_number] = 1;    break;}
  case 27: {clear_cell[cell_number] = 0;      break;}
  case 28: {priority[cell_number] = 0;        break;}
  case 39: {if(local_state[cell_number] == 0xc)
           local_state[cell_number] = 1;    break;}
}}
if(clear_cell[cell_number] != 0)
  fault_register[cell_number] = fault_register[cell_number] | 0x1000;
else fault_register[cell_number] = fault_register[cell_number] & 0xefff;
}
/*************************************************************************
      debug only next five lines
*************************************************************************/
if((GetUserInt() == 0) && (local_state[cell_number] != 0)){
printf("state of cell %d changed from 0 to %d\n",cell_number,
                  local_state[cell_number]);
if((global_state[cell_number] != 0))
  local_state[cell_number] = 0;   }
/*************************************************************************
*
          BEGIN EXTERNAL CONNECTION ALGORITHM
*
*************************************************************************/
if((GetArcInt(OUT0) != 1)){
if((local_state[cell_number] != 0)){
if((fault_register[cell_number] & 1) == 1)
{      enable_1[cell_number] = 0;
  if(disable_0[cell_number] == 0)    enable_0[cell_number] = 1;
  else enable_0[cell_number] = 0;    }
else {  enable_1[cell_number] = 1;        enable_0[cell_number] = 0; }

if((fault_register[cell_number] & 0x40) != 0x40)
{      enable_6[cell_number] = 1;        enable_5[cell_number] = 0;}

else {  enable_6[cell_number] = 0;        enable_5[cell_number] = 1;   }
```

```
enable_2[cell_number] = 0; enable_3[cell_number] = 0;
enable_4[cell_number] = 0; enable_8[cell_number] = 0;
enable_7[cell_number] = 0; enable_9[cell_number] = 0;

fault_register_north[cell_number] =
        (fault_register[cell_number] & 0x15b);
fault_register_south[cell_number] =
        (fault_register[cell_number] & 0x26d);
if((fault_register[cell_number] & 0x0c0) != 0)
 { fault_register_north[cell_number] =
        (fault_register_north[cell_number] | 0x40);
  fault_register_south[cell_number] =
        (fault_register_south[cell_number] | 0x40);
  }
/* DECODE THE CONTENTS OF THE FAULT-REGISTER                    */
switch (fault_register_north[cell_number] )    {
   case 0:   { enable_3[cell_number] = 1; break;}
   case 1:   { enable_2[cell_number] = 1; break;}
   case 2:   { enable_4[cell_number] = 1; break;}
   case 3:   { enable_2[cell_number] = 1; break;}
   case 8:   { enable_2[cell_number] = 1; break;}
   case 9:   { enable_2[cell_number] = 1; break;}
   case 10:  { enable_2[cell_number] = 1; break;}
   case 11:  { enable_2[cell_number] = 1; break;}
   case 16:  { enable_4[cell_number] = 1; break;}
   case 17:  { enable_3[cell_number] = 1; break;}
   case 18:  { enable_4[cell_number] = 1; break;}
   case 19:  { enable_2[cell_number] = 1; break;}
   case 24:  { enable_3[cell_number] = 1; break;}
   case 25:  { enable_3[cell_number] = 1; break;}
   case 26:  { enable_2[cell_number] = 1; break;}
   case 27:  { enable_2[cell_number] = 1; break;}
   case 64:  { enable_3[cell_number] = 1; break;}
   case 65:  { enable_3[cell_number] = 1; break;}
   case 66:  { enable_4[cell_number] = 1; break;}
   case 67:  { enable_4[cell_number] = 1; break;}
   case 72:  { enable_3[cell_number] = 1; break;}
   case 73:  { enable_3[cell_number] = 1; break;}
   case 74:  { enable_4[cell_number] = 1; break;}
   case 75:  { enable_4[cell_number] = 1; break;}
   case 80:  { enable_4[cell_number] = 1; break;}
   case 81:  { enable_4[cell_number] = 1; break;}
   case 82:  { enable_4[cell_number] = 1; break;}
   case 83:  { enable_4[cell_number] = 1; break;}
   case 88:  { enable_4[cell_number] = 1; break;}
   case 89:  { enable_4[cell_number] = 1; break;}
   case 90:  { enable_4[cell_number] = 1; break;}
   case 91:  { enable_4[cell_number] = 1; break;}
   case 256: { enable_3[cell_number] = 1; break;}
   case 257: { enable_2[cell_number] = 1; break;}
   case 258: { break;}
   case 259: { enable_2[cell_number] = 1; break;}
   case 264: { enable_2[cell_number] = 1; break;}
   case 265: { enable_2[cell_number] = 1; break;}
   case 266: { enable_2[cell_number] = 1; break;}
   case 267: { enable_2[cell_number] = 1; break;}
```

```
  case 272: { enable_3[cell_number] = 1; break;}
  case 273: { enable_2[cell_number] = 1; break;}
  case 274: { break;}
  case 275: { enable_2[cell_number] = 1; break;}
  case 280: { enable_2[cell_number] = 1; break;}
  case 282: { enable_2[cell_number] = 1; break;}
  case 281: { enable_2[cell_number] = 1; break;}
  case 283: { enable_2[cell_number] = 1; break;}
  case 320: { enable_3[cell_number] = 1; break;}
  case 321: { enable_3[cell_number] = 1; break;}
  case 322: { break;}
  case 323: { break;}
  case 328: { enable_3[cell_number] = 1; break;}
  case 329: { enable_4[cell_number] = 1; break;}
  case 330: { break;}
  case 331: { break;}
  case 336: { enable_3[cell_number] = 1; break;}
  case 337: { enable_3[cell_number] = 1; break;}
  case 338: { break;}
  case 339: { break;}
  case 344: { enable_3[cell_number] = 1; break;}
  case 345: { enable_3[cell_number] = 1; break;}
  case 346: { break;}
  case 347: { break;}
    default:{
printf("no north match found   cell = %d, register =%d  \n",
                    cell_number,fault_register[cell_number]);}
  }
 switch (fault_register_south[cell_number] )    {
  case 0:   { enable_8[cell_number] = 1; break;}
  case 1:   { enable_9[cell_number] = 1; break;}
  case 4:   { enable_7[cell_number] = 1; break;}
  case 5:   { enable_9[cell_number] = 1; break;}
  case 8:   { enable_9[cell_number] = 1; break;}
  case 9:   { enable_9[cell_number] = 1; break;}
  case 12:  { enable_9[cell_number] = 1; break;}
  case 13:  { enable_9[cell_number] = 1; break;}
  case 32:  { enable_7[cell_number] = 1; break;}
  case 33:  { enable_8[cell_number] = 1; break;}
  case 36:  { enable_7[cell_number] = 1; break;}
  case 37:  { enable_9[cell_number] = 1; break;}
  case 40:  { enable_8[cell_number] = 1; break;}
  case 41:  { enable_8[cell_number] = 1; break;}
  case 44:  { enable_9[cell_number] = 1; break;}
  case 45:  { enable_9[cell_number] = 1; break;}
  case 64:  { enable_8[cell_number] = 1; break;}
  case 65:  { enable_8[cell_number] = 1; break;}
  case 68:  { enable_7[cell_number] = 1; break;}
  case 69:  { enable_7[cell_number] = 1; break;}
  case 72:  { enable_8[cell_number] = 1; break;}
  case 73:  { enable_8[cell_number] = 1; break;}
  case 76:  { enable_7[cell_number] = 1; break;}
  case 77:  { enable_7[cell_number] = 1; break;}
  case 96:  { enable_7[cell_number] = 1; break;}
  case 97:  { enable_7[cell_number] = 1; break;}
  case 100: { enable_7[cell_number] = 1; break;}
```

```
   case 101: { enable_7[cell_number] = 1; break;}
   case 104: { enable_7[cell_number] = 1; break;}
   case 105: { enable_7[cell_number] = 1; break;}
   case 108: { enable_7[cell_number] = 1; break;}
   case 109: { enable_7[cell_number] = 1; break;}
   case 140: { enable_7[cell_number] = 1; break;}
   case 512: { enable_8[cell_number] = 1; break;}
   case 513: { enable_9[cell_number] = 1; break;}
   case 516: { break;}
   case 517: { enable_9[cell_number] = 1; break;}
   case 520: { enable_9[cell_number] = 1; break;}
   case 521: { enable_9[cell_number] = 1; break;}
   case 524: { enable_9[cell_number] = 1; break;}
   case 525: { enable_9[cell_number] = 1; break;}
   case 544: { enable_8[cell_number] = 1; break;}
   case 545: { enable_9[cell_number] = 1; break;}
   case 548: { break;}
   case 549: { break;}
   case 552: { enable_9[cell_number] = 1; break;}
   case 553: { enable_9[cell_number] = 1; break;}
   case 556: { break;}
   case 557: { enable_9[cell_number] = 1; break;}
   case 576: { enable_8[cell_number] = 1; break;}
   case 577: { enable_8[cell_number] = 1; break;}
   case 580: { break;}
   case 581: { break;}
   case 584: { enable_8[cell_number] = 1; break;}
   case 585: { enable_8[cell_number] = 1; break;}
   case 588: { enable_8[cell_number] = 1; break;}
   case 589: { break;}
   case 608: { enable_8[cell_number] = 1; break;}
   case 609: { enable_8[cell_number] = 1; break;}
   case 612: { break;}
   case 613: { break;}
   case 616: { enable_8[cell_number] = 1; break;}
   case 617: { enable_8[cell_number] = 1; break;}
   case 620: { break;}
   case 621: { break;}
     default:{
 printf("no south match found   cell = %d, register =%d  \n",
                    cell_number,fault_register[cell_number]);}
     }

 /*********************** end of case statements ********************** */

/*****************************************************************************
      LOGIC FOR CELLS BORDERING A MULTIPLE FAULT CONDITION
*****************************************************************************/

if (double_fault_north[cell_number] != 0) {
  double_fault[cell_number] = double_fault[cell_number] | 8;
  SetArcColor(OUT3,BLACK); enable_3[cell_number] = 0;
   if ((((fault_register[cell_number] & 0x9)!= 0)&&
         ((fault_register[cell_number] & 0xc0) != 0))
    { ++double_fault_north_delay[cell_number];
```

```
    if((double_fault_north_delay[cell_number] < 4))
       enable_2[cell_number] = 1;  } }

if( double_fault_east[cell_number] != 0){
 double_fault[cell_number] = double_fault[cell_number] | 2;
   enable_5[cell_number] = 0; enable_6[cell_number] = 0;
  SetArcColor(OUT5,BLACK); SetArcColor(OUT6,BLACK);}

if (double_fault_west[cell_number] != 0){
 double_fault[cell_number] = double_fault[cell_number] | 1;
 enable_1[cell_number] = 0;
 if(double_fault_west[cell_number] > 4)
          enable_0[cell_number] = 0;
else { if(disable_0[cell_number] == 0)    enable_0[cell_number] = 1;
       SetArcColor(OUT0,BLACK);
       SetArcColor(OUT1,BLACK); }}

if(double_fault_south[cell_number] != 0){
 double_fault[cell_number] = double_fault[cell_number] | 4;
    SetArcColor(OUT8,BLACK); enable_8[cell_number] = 0;
  if (((fault_register[cell_number] & 0x9)!= 0)&&
         ((fault_register[cell_number] & 0xc0) != 0))
   {   ++double_fault_south_delay[cell_number];

    if((double_fault_south_delay[cell_number] < 5))
       enable_9[cell_number] = 1;  }
}
double_fault1[cell_number] = double_fault[cell_number];
if(double_fault_northwest[cell_number] != 0)
{
      enable_2[cell_number] = 0;
      double_fault1[cell_number] = 3;

/*************************************************************************
 DELAY IMPLIMENTATION OF WEST OR FAR WEST FAULT CONNECTION IN
ORDER TO COMMUNICATE TO THE NORTH ROW- ELSE NORTH ROW WILL NOT KNOW
OF THE FAULT TO THE FAR WEST IN THE SOUTH ROW BECAUSE OF NO COMMUNICATIONS
TO THE WEST OF THE DOUBLE_FAULT-  THIS INFORMATION MUST BE TRANSMITTED BY
DOUBLE_FAULT_NORTHWEST CELL
*************************************************************************/

 if((fault_register[cell_number] & 0x9) != 0)
  {  ++double_fault_northwest[cell_number];
    if(double_fault_northwest[cell_number] < 5){
      if((fault_register[cell_number] & 0x2) != 0)
         enable_4[cell_number] = 1;
      else  enable_3[cell_number] = 1;}
    } }
if(double_fault_northeast[cell_number] != 0)
{
      enable_4[cell_number] = 0;
      double_fault1[cell_number] = 7;
}
if(double_fault_southwest[cell_number] != 0)
{     double_fault1[cell_number] = 5;
      enable_9[cell_number] = 0;
```

```
/*************************************************************************
DELAY IMPLIMENTATION OF WEST OR FAR WEST FAULT CONNECTION IN
ORDER TO COMMUNICATE TO THE SOUTH ROW- ELSE SOUTH ROW WILL NOT KNOW
OF THE FAULT TO THE FAR WEST IN THE NORTH ROW BECAUSE OF NO COMMUNICATIONS
TO THE WEST OF THE DOUBLE_FAULT.  THIS INFORMATION MUST BE TRANSMITTED BY
DOUBLE_FAULT_SOUTHWEST CELL
*************************************************************************/

 if((fault_register[cell_number] & 0x9) != 0)
  {  + +double_fault_southwest[cell_number];
    if(double_fault_southwest[cell_number] < 5){
  if((fault_register[cell_number] & 0x4) != 0)
     enable_7[cell_number] = 1;
  else  enable_8[cell_number] = 1;}
    } }
if(double_fault_southeast[cell_number] != 0)
{
      enable_7[cell_number] = 0;
      double_fault1[cell_number] = 6;
}}
SetArcInt(OUT1,fault_register[cell_number]& 0x3ff);

/*************************************************************************
       BLACK OUT EVERYTHING FOR FAULTY CELLS
*************************************************************************/

  if(local_state[cell_number] = = 0){
    SetArcColor(OUT0,BLACK);
    SetArcColor(OUT1,BLACK);
    SetArcColor(OUT2,BLACK);
    SetArcColor(OUT3,BLACK);
    SetArcColor(OUT4,BLACK);
    SetArcColor(OUT5,BLACK);
    SetArcColor(OUT6,BLACK);
    SetArcColor(OUT7,BLACK);
    SetArcColor(OUT8,BLACK);
    SetArcColor(OUT9,BLACK);
/* ALL PORTS ARE ACTIVATED ON FAULTY CELLS                    */
  if(PortIsActive(IN10) = = 1) {
    ActivatePort(OUT0); ActivatePort(OUT1); ActivatePort(OUT2);
    ActivatePort(OUT3); ActivatePort(OUT4); ActivatePort(OUT5);
    ActivatePort(OUT6); ActivatePort(OUT7); ActivatePort(OUT8);
    ActivatePort(OUT9);}
    enable_0[cell_number] = 0;  enable_1[cell_number] = 0;
    enable_2[cell_number] = 0;  enable_3[cell_number] = 0;
    enable_4[cell_number] = 0;  enable_5[cell_number] = 0;
    enable_6[cell_number] = 0;  enable_7[cell_number] = 0;
    enable_8[cell_number] = 0;  enable_9[cell_number] = 0;
    out0[0] = 0; out1[0] = 0; out2[0] = 0; out3[0] = 0; out4[0] = 0;
    out5[0] = 0; out6[0] = 0; out7[0] = 0; out8[0] = 0; out9[0] = 0;
             }

}
/*************************************************************************
*
```

```
            BEGIN EXTERNAL CONNECTIONS
*
********************************************************************** */

/**************************************************************************
    OUTPUT DATA IS GATED TO THOSE PORTS THAT ARE ACTIVE, THE PORTS ARE
    ACTIVATED,  ... ALL OTHERS LEFT BLACK AND DEACTIVATED
**************************************************************************/

 if(local_state[cell_number] != 0){
   {
      if(enable_0[cell_number] == 1){
        SetArcColor(17,BLACK);  SetArcColor(16,RED);
if((GetNTimes() != 1) && (PortIsActive(10) == 1)) ActivatePort(OUT0);
    {  out1[0] = 0xd;
       out0[0] = output_west ;}
    }

  else    if (enable_1[cell_number] == 1)
   {  SetArcColor(16,BLACK);  SetArcColor(17,RED);
if((GetNTimes() != 1) && (PortIsActive(10) == 1))  ActivatePort(OUT1);
      { out0[0] = 0xd;  out1[0] = output_west;}     }
else {SetArcColor(16,BLACK);  SetArcColor(17,BLACK); }

      if(enable_5[cell_number] == 1){
        SetArcColor(22,BLACK);  SetArcColor(21,RED);
if((GetNTimes() != 1) && (PortIsActive(10) == 1))   ActivatePort(OUT5);
      {  out5[0] = output_east; out6[0] = 0xd; }          }

    else if (enable_6[cell_number] == 1)
{  SetArcColor(21,BLACK);  SetArcColor(22,RED);
if((GetNTimes() != 1) && (PortIsActive(10) == 1))  ActivatePort(OUT6);
  if((reconfigure[cell_number] == 1) && (GetArcInt(OUT0) == 2))
      out6[0] = output_east | 0x200000;
   else   { out6[0] = output_east ; out5[0] = 0xd; }          }
else {SetArcColor(21,BLACK);  SetArcColor(22,BLACK); }

/**********CODE FOR PORTS TO THE NORTH  ***********************************/
      if(enable_2[cell_number] == 1){
   SetArcColor(18,RED); SetArcColor(19,BLACK); SetArcColor(20,BLACK);
if((GetNTimes() != 1) && (PortIsActive(10) == 1)) ActivatePort(OUT2);
      {  out2[0] = output_north; out3[0] = 0xd; out4[0] = 0xd;}
       }

   else if(enable_3[cell_number] == 1){
    SetArcColor(18,BLACK); SetArcColor(19,RED); SetArcColor(20,BLACK);
if((GetNTimes() != 1) && (PortIsActive(10) == 1)) ActivatePort(OUT3);
      {  out2[0] = 0xd; out3[0] = output_north; out4[0] = 0xd;}
                  }

  else   if (enable_4[cell_number] == 1){
    SetArcColor(18,BLACK); SetArcColor(19,BLACK); SetArcColor(20,RED);
if((GetNTimes() != 1) && (PortIsActive(10) == 1)) ActivatePort(OUT4);
      {  out2[0] = 0xd; out3[0] = 0xd; out4[0] = output_north;}
                  }
else {SetArcColor(18,BLACK);  SetArcColor(19,BLACK); SetArcColor(20,BLACK);}
```

```
/*********CODE FOR PORTS TO THE SOUTH  ************************************/

      if(enable_7[cell_number] == 1) {
    SetArcColor(23,RED); SetArcColor(24,BLACK); SetArcColor(25,BLACK);
if((GetNTimes() != 1) && (PortIsActive(10) == 1)) ActivatePort(OUT7);
      { out7[0] = output_south; out8[0] = 0xd; out9[0] = 0xd;}
                    }

else   if(enable_8[cell_number] == 1) {
    SetArcColor(23,BLACK); SetArcColor(24,RED); SetArcColor(25,BLACK);
        if((GetNTimes() != 1) && (PortIsActive(10) == 1)) ActivatePort(OUT8);
      { out7[0] = 0xd; out8[0] = output_south; out9[0] = 0xd;}
                }

   else if(enable_9[cell_number] == 1){
    SetArcColor(23,BLACK); SetArcColor(24,BLACK); SetArcColor(25,RED);
    if((GetNTimes() != 1) && (PortIsActive(10) == 1)) ActivatePort(OUT9);
      { out7[0] = 0xd; out8[0] = 0xd; out9[0] = output_south;}
          }
else {SetArcColor(23,BLACK);  SetArcColor(24,BLACK); SetArcColor(25,BLACK);}
}}

/*****************************************************************************
*
              END OF THE CONNECTION ALGORITHM
*
*****************************************************************************/

/* REMAINING CODE IS FOR DEBUG AND TEST USE ONLY  NOT OPERATIONAL CODE  */
enable_register[cell_number] = 0;
if(enable_1[cell_number] == 1) enable_register[cell_number] = 1;
else if (enable_0[cell_number] != 1) enable_register[cell_number] = 9;
if(enable_2[cell_number] == 1) enable_register[cell_number] =
                        enable_register[cell_number] + 20;
else if(enable_3[cell_number] == 1)enable_register[cell_number] =
                        enable_register[cell_number] + 30;

else if(enable_4[cell_number] == 1)enable_register[cell_number] =
                        enable_register[cell_number] + 40;

if(enable_5[cell_number] == 1) enable_register[cell_number] =
                        enable_register[cell_number] + 500;

else if(enable_6[cell_number] == 1) enable_register[cell_number] =
                        enable_register[cell_number] + 600;

if(enable_7[cell_number] == 1)enable_register[cell_number] =
                        enable_register[cell_number] + 7000;

else if(enable_8[cell_number] == 1) enable_register[cell_number] =
                        enable_register[cell_number] + 8000;

else if(enable_9[cell_number] == 1) enable_register[cell_number] =
                        enable_register[cell_number] + 9000;
```

```
SetArcInt(OUT9,enable_register[cell_number]);
/*NODE WILL GET COLOR OF LOCAL STATE IF EITHER 1 AND NO GLOBAL STATE(QUIESCENT)
  OR NON QUIESCENT CELL, OR THE COLOR OF THE GLOBAL STATE IF CELL HAS SEED  */

if((global_state[cell_number] != 0) && (local_state[cell_number] == 1))
   SetNodeColor(global_state[cell_number]);

else SetNodeColor(local_state[cell_number]);

}
```

The vita has been removed from the scanned document